Master Thesis on Sound and Music Computing

Universitat Pompeu Fabra

Neural Texture Sound Synthesis with physically-driven continuous controls using synthetic-to-real unsupervised Domain Adaptation

Matteo Fabbri

Supervised by: Lonce Wyse

August 2023

1. 

1. Introduction

Texture sounds are ever-present in our daily lives, and form a crucial part of media sound design. In spite of their ubiquitousness, there has not yet been a comprehensive adoption of automated synthesis methods, unlike music and speech; very often, texture sounds are manually recorded and edited for media usage.
The primary focus of this research is to build some Data-driven supervised Deep Learning models for Sound Texture synthesis affording some high-level, continuous and physically-driven user interaction.
For example, by continuous controls we refer to values controllable with a knob or a slider, rather than with a button. By physically-driven synthesis parameters, we indicate some characteristics of the objects involved in the production of the sounds in the real world (e.g. some ‘rain’ sounds could be synthesised by controlling the amount of rain, as well as the material over which the rain falls). The main evaluation test for the generated sounds is how close the system can reproduce the quality of real texture sounds.

1.1 Audio textures

(Saint-Arnaud & Popat, 1995) first defined Audio textures as having constant long term characteristics and an attention span -in Machine Learning jargon, a similar concept is called *time dependency*-, that is the maximum amount of time between events before they are perceived as distinct, of at most a few seconds. In fact, the amount of new information over time, in a cognitive sense, is much larger in music and speech than audio textures. Textures, on the other hand, carry more new information over time than noise, with which share the characteristic of being based on statistics, much more than music and speech. In other words, the probability distribution of music and speech varies to a greater extent over time than it does on texture sounds, which tend to be more stable at larger timescales.

Figure 1.1 Potential information content over time between noise, sound textures and music/speech. Figure reproduced from (Saint-Arnaud & Popat, 1995).

More specifically, texture sounds refer usually to a compound made of small parts building up to a whole, consisting in low-level atomic elements and higher-level elements describing the distribution of atoms over time. Atoms can have a hierarchical structure of periodicity, stochasticity, co-occurrence or sequence-like; each single atom may not be always singularly perceivable or fundamental, but nonetheless the overall sound is different, both perceptually and statistically, from noise (Huzaifah Bin Md Shahrin, 2020, p. 55).
For simplicity, a distinction is made between homogeneous sound textures and soundscapes; the latters are combinations of the former, which can be more easily modelled and synthesised. Soundscapes can nevertheless be synthesised by combining two or more homogeneous textures.
The varied and somehow unstructured nature of texture sounds makes their modelling complex and challenging.

1.2 Challenges in Neural Audio synthesis

Noticeably, Audio has high-dimensionality, that is, representing good-quality Audio needs a large amount of data (e.g. usually good-quality Audio is encoded at 16 bit per sample, and sampled at 44.1 kHz per second). Any time dependency-related problem builds on this aspect, as the amount of data to keep track of increases as the considered time span increases. Furthermore, the complex phase structure of the Audio makes raw waveform representations of Audio inadequate for any perceptually-relevant comparison. In other words, differences between two audio waveforms at the sample level may not be perceived at all; from this, it is hence also

true that a number of waveforms looking different, may sound exactly the same to the human ear. As we will see, we can not then define loss functions (in a Supervised Learning set-up, at training stage, a loss function, or simply loss, is any metric used to evaluate how distant the output of a model is from the ground truth). While this problem can be addressed by using frequency representations, which is much more perceptually-relevant, power-only-spectrum representations (usually used for fast synthesis) are a lossy representation and Audio synthesis can then present low-quality.

1.3 Challenges in Neural Texture Sound synthesis

Despite recent advancements in Generative Deep Learning models, no particular attention has been put on texture sounds synthesis; most of the work done on environmental sounds, inevitably smaller than the one on music and speech, is for acoustic scene classification.
This is both cause and consequence of the lack of big and well-labelled audio datasets oriented to synthesis rather than classification.
Ordinal, continuous labels representing physically-based control parameters (e.g. ‘amount of water’ in a container filled with water sound), are in fact hard to obtain. Furthermore, since different sounds require radically different control parameters (e.g. a rain sound might have an ‘amount of rain’ control, while a wind sound has other completely different controls), the synthesis and control parts of a system heavily depend, influence and constrain each other (Huzaifah Bin Md Shahrin, 2020, p. 5). Nevertheless, a still-open research question is how to deal with long time dependencies with Deep networks, and also how to marry this aspect with low-memory and high inference speed requirements for real-time systems.

1.4 Who would benefit

There is an increasing demand, especially in settings like media production, for more adaptable real-time systems. Specifically, the interactive media sector (Videogames, Virtual Reality, and Augmented Reality), as immersivity becomes more and more important, would benefit from audio synthesis models that automatically generate sounds reacting to some user input or game parameters, without relying on audio files playback.

Nowadays, in fact, most of the interactive entertainment Industry still use large amounts of pre-recorded audio files, which are memory expensive and non-dynamic; audio looping also goes against user immersion and realism.
Non-interactive media productions like sound designers and composers for TV/Cinema can also benefit from intuitively controllable sound texture synthesis, also but not exclusively by leveraging the creative possibilities offered by Deep Learning in latent space navigation for creating new sounds, not present in the training dataset. In fact, Generative models, in addition to generating sounds similar to the ones present in the training set, can create, by interpolation, the in-between of these sounds and afford novel sonic scenarios.

2. Deep learning background

2.1 Generative Neural Network models

Generative Neural Network models, unlike discriminative models, are tightly bonded with, and can be better investigated if considered through the lenses of Probability Theory.
The main goal of a Generative Neural Network model is to approximate a *prior* Probability Distribution, from which the training examples are drawn. In other words, a Generative Neural Network learns to replicate the latent statistics behind observed data.
A common method to measure the difference between the *conditional* distribution generated by the model, and the prior distribution of the training data-set, is the Kullback-Leibler (KL) divergence (Huzaifah & Wyse, 2020). Minimising the KL divergence means to maximise the similarity between conditional and prior distributions (maximum likelihood estimation).
Specifically, at the light of likelihood representation, there are two main types of Generative Neural Networks; networks that allow for the *explicit* formulation of the marginal likelihood p(X) of the data (e.g. autoregressive networks and variational autoencoders), and networks having *implicit* knowledge of the underlying probability distribution to sample from (e.g. GAN). In practice, being able to calculate the likelihood provides an unambiguous way of measuring how close the model is in approximating the real data distribution; with explicit knowledge of training data likelihood, we can generally obtain better evaluation metrics (Li et al., 2017), while with implicit

knowledge, we can only evaluate the model by looking at the generated examples and comparing them to the real data (Huzaifah & Wyse, 2020).

2.1.1 Auto-regressive models

Autoregressive-types of NN model the following formula, for a tractable (thus, directly optimizable) density function.

$$p(x) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1})$$

Figure 2.1.1 The probability of the next element to occur, is equal to the product of all past events' probabilities.

In other words, this is a causal system where the current term in the sequence is conditioned only on past events. This applies to RNN and particular types of CNN like WaveNet (van den Oord et al., 2016). In CNN, simpler to train than RNN, Convolutional Layers are used to process a sequence of audio samples; specifically, dilation is used to increase the temporal dependency capabilities -represented, in CNN, with the receptive field- of the network, without increasing its size. The dilation factor increases exponentially with layers (outermost layers have higher dilation).

2.2 Generative Neural Network architectures

2.2.1 Convolutional Neural Networks

Initially developed for analysing visual imagery, Convolutional Neural Networks -CNN- are based on the concept of space-invariant filters (also called kernels), which are convoluted and 'slide' over the input image in order to detect patterns. A common terminology in Deep Learning jargon is 'receptive field'; that is, the area of the input image that a particular neuron in the network "sees" through its filters. While the number of filters (usually, as many as the features to be learned from the data) is an hyperparameter of the network, the content of the filters theirselves is learned during training. Depending on the number of dimensions of the input data (and on the number of dimensions across which the Convolutional filters are moved, or 'strided'), the Convolutional filters do not necessarily apply 2D convolutions; 1D and 3D convolutions are also available.

For Audio, 1D convolutions are applied to the raw-audio waveform in end-to-end set-ups, while 2D convolutions are applied to the FFT-based spectrogram of the Audio. CNN can not only be used for Audio analysis and MIR tasks, but for Audio synthesis as well. After each Convolutional layer, in fact, a Pooling layer is usually present, in order to down-sample the data into a higher-level representation with fewer number of dimensions, or, conversely, an Unpooling layers is present, in order to up-sample the data into a lower level of abstraction with fewer number of dimensions. Thus, Pooling is performed during encoding, Unpooling is performed during decoding (synthesis). Specifically for Audio Synthesis, WaveNet (van den Oord et al., 2016), an autoregressive and causal CNN for generating Audio at the raw waveform level. To address long-range temporal dependencies, dilated causal convolutions, which provide larger receptive fields, have been used. A dilated convolution is a convolution where the filter is applied to a longer sequence, by skipping input values according to a defined step, called ‘dilation factor’ (which increases exponentially in the original paper implementation). Stacked dilated convolutions enable networks to have larger receptive fields with fewer layers.
There are no pooling layers in WaveNet because the model is designed to generate raw audio waveforms, which require a high level of temporal resolution. Pooling layers reduce the temporal resolution of the output, which can lead to loss of information and artefacts in the generated audio. Instead, WaveNet uses dilated convolutions to increase the receptive field of the model, as well as enlarge its time dependencies, while maintaining the temporal resolution of the output. Predictions are sequential: after each sample is predicted, it is fed back into the network to predict the next sample. Furthermore, it is important to note that WaveNet implements a classification task rather than a regression task; softmax is a common activation function for multi-class classification problems. The model is trained to predict the probability distribution of the next audio sample given the previous samples; this can be framed as a multi-class classification problem, where each class corresponds to a possible value of the next audio sample.

(Dieleman & van den Oord, 2018) created hierarchical WaveNets in order to learn longer time dependencies for raw-waveform Music generation. The model decouples the learning of local and larger-scale structures by training separate WaveNet models for each level of abstraction. Each WaveNet is trained as a decoder in an autoencoder setup using a Vector-Quantized Variational Auto Encoder -VQ-VAE-, which admits a

discretised feature space. The feature space learned is a compressed encoding containing information on the high-level structure of the input. By discretizing the latent space into a finite number of codebooks, the VQ-VAE can learn a more compact representation of the input data, which can improve the efficiency of data compression and generation. Additionally, the use of discrete codebooks allows for greater control over the properties of the generated data.
It is then used to condition another VQ-VAE working at a lower level. The hierarchy can be extended to an arbitrary number of VQ-VAE tiers.

(Ramires, 2023, pag. 96) used Wave-U-Net (Stoller et al., 2018) rather than WaveNet for synthesis of one-shot percussive sounds since long time dependencies are not needed for this type of sounds.

(Esling et al., 2023) used 1D Convolutional layers in both encoder and decoder of RAVE in order to achieve very fast inference speed for timbre transfer applications.

2.2.2 Recurrent Neural Networks

Whereas convolutional layers contain weights that are shared across “spatial” dimensions, recurrent layers have weights that are shared across timesteps. RNNs excel at capturing patterns in sequences of data, such as audio samples. Recurrent layers are so called because they perform the same processing steps on each element in sequence, while maintaining an internal memory of previous steps called the “hidden state”.
Like CNN assume some space consistency, RNN assume some time consistency, so that the weights are shared across time (in fact, if parameters are not shared, we would obtain a regular Feedforward network rather than a RNN). This characteristic allows RNN to accept and produce any arbitrary input and output length (sequence to sequence, where input and output sequences have different length, is also possible using an encoder-decoder Architecture, where the latent representation is a fixed-length sequence of the variable-length input sequence), and to substantially decrease the number of trainable parameters. In fact, while reading a sequence, if a RNN model uses different parameters for each step during training, even though the encoding of different input/output length can be achieved with padding, it won't generalise to unseen sequences of different lengths. Furthermore, in many domains, the sequences tend to operate according to the same rules, regardless of the time

step, that is, at each time step. Like feature maps are location independent and transform-invariant in CNN (we can assume that there are similar interesting patterns in different regions of the picture, and at any scale, rotation and translation), parameters are time-independent in RNN, meaning that we can assume to find patterns at different time steps, regardless of when they occur.
In fact, as CNN filters are ‘slide’ across the input, RNN layers are also replicated over different time steps of the same input.
Backpropagation for RNN (also called BPTT -Backpropagation through time-), since the weights are the same across different time steps, is performed by accumulating the contributions of the parameters gradients at each time step.
The drawback of parameters sharing is that the model applies the same transformation to the input at every time step, and it thus needs to learn a transformation that makes sense for the entire sequence. Also, another issue of RNNs is the long time they take to generate a complete sequence, due to the networks only being able to generate one point at a time because of the explicit conditioning on previous points.
Furthermore, since RNN are practically copies of the same network, they can become very deep and subject to the vanishing gradient problem, for which weights in layers very far from the start of the backpropagation get updated by no or extremely low amount, which effectively results in no learning for longer time dependencies which go further back in time. This inherent characteristic of RNN brought to the appearance of Long short-term memory networks -LSTM- and Gated Recurrent Unit -GRU-, a specialisation of RNN which try to address the vanishing gradient problem by selecting what in the history of past predictions to memorise, and what to forget.
Other innovations to further increase the receptive field include hierarchically organised RNNs, like the one used by (Huzaifah Bin Md Shahrin, 2020) in audio texture synthesis.
Automatic learning of timescale boundaries have been researched, where in a hierarchical multi-tier RNN each timescale is modelled by a different tier; a network layer feeds a summarised representation of the input segment into the next layer whenever the boundary detector is turned on during a timestep, indicating what the model thinks is the end of a segment corresponding to the level of abstraction of that layer.

2.2.3 Variational Auto-Encoders

Variational Autoencoders (VAE) model the following intractable (due to the presence of an integral, and not directly optimizable) density function:

$$p_\Theta(x) = \int p_\Theta(z) p_\Theta(x|z) dz$$

Figure 2.2.3 The probability density function modelled by a VAE, where z (latent variables) has smaller size than x, as Autoencoders capture meaningful features in the training data by performing dimensionality reduction.

Technically, Autoencoders are an unsupervised learning technique; in practice, even though no labels are used, the very same input data is used as ground truth, in order for the decoder network to approximately reconstruct the input data. The assumption beyond Autoencoders is that the training data is generated from some underlying unobserved latent representation z, where z is a vector where each element is a high-level feature capturing the amount of some factor of variation (e.g. amount of 'smile' in a face image) that we have in our training data. While the prior probability distribution p(z) is usually chosen to be a Gaussian distribution, the conditional distribution p (x | z) (where we sample our generated data from) is complex and we model it with a decoder Neural Network (Li et al., 2017).
To make the equation in figure 2.2 tractable and optimizable, we also create an encoder q (z | x) that approximates p (z | x). In practical terms, to fully represent the dataset distribution, z is represented with two vectors, one for the mean and one for the standard deviation.
In simpler terms, in Variational Autoencoders, the encoder produces a distribution over z (prior), the decoder produces a distribution over x (posterior); we sample from both distributions and we try to minimise the divergence between posterior and prior, by maximising the likelihood of the original input being reconstructed at the output. This divergence is measured with the reconstruction loss.
Furthermore, the latent space is forced to resemble as much as possible the prior distribution, usually represented with a normal distribution; this process is trained using a regularisation loss. Another aim of the regularisation loss is to make sure that the model that does not overfit (likely scenario when a large number of parameters is present) to the training dataset, that is, the model should not poorly generalise by just

learning to copy the input to the output (in which case, the Auto Encoder would not be Variational). A way of implementing regularisation is sharing the parameters between Encoder and Decoder, a method called tied-weights.

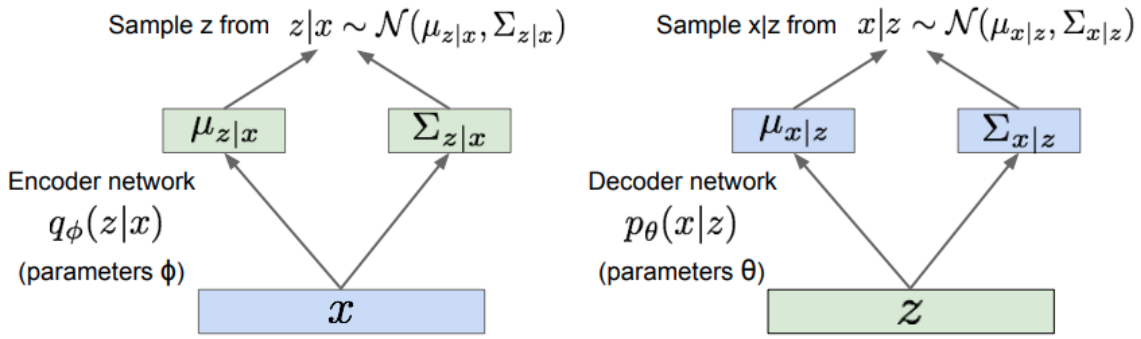


Figure 2.2.4 Diagram of a Variational Auto Encoder. Figure reproduced from (Li et al., 2017).

Usually, the reconstruction quality of Auto Encoders is not excellent, and GANs are used in conjunction of Auto Encoders in order to address this problem, like in RAVE (Esling et al., 2021).

2.2.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs) do not work with any explicit density function, but rather learn to transform a starting random noise distribution into the probability density function of the training set, as a minimax two-player game. The two Networks are a Generator and a Discriminator, and the objective is that, at inference time (when ideally, but hardly in practice, a state of equilibrium has been reached), the generator (re)produces perfectly the true data distribution, which ultimately leads to the discriminator merely randomly guessing the input label, unable to tell the real from the fake. While the discriminator is exposed to both real and modelled data, the generator has no direct access to the training samples, and so has to develop an understanding of the data distribution solely via the error signal provided by the discriminator.
Since the GAN framework does not impose any restriction on the structure of z, the Generator may use it in a highly entangled way, where individual dimensions do not correspond to high-level, humanly meaningful features of the training data.
One way to induce additional structure is to use a conditioning strategy, either as providing both Generator and Discriminator some extra information, or decomposing z into two parts: a noise vector like before, and a latent code that is trained to have high

mutual information with the generator distribution. This latent code can be used to discover features in an unsupervised fashion (Huzaifah & Wyse, 2020).

2.2.5 Transformer-based Architectures

Transformer-based Architectures have self-attention as their fundamental mechanism. Self-attention, which is the only operation that put any of the input elements in relation to each other as all other operations are performed separately on input elements, is a (variable length) sequence-to-sequence operation; for each element in the input sequence, an element in the output sequence is calculated as the dot product of the input sequence and a weight sequence (which sums up to 1 over the whole sequence, and is derived from the input sequence itself). So, each output element is a weighted sum over an input sequence.

$$y_i = \sum_j w_{ij} x_j$$

Figure 2.2.5 Formula for the self-attention mechanism; z is the input sequence, w is the set of weights, and y is the output sequence.

The outputs are then mapped between 0 and 1, and they also sum up to 1 over the whole sequence. The most interesting aspect of self-attention is probably the way the weights (w) are calculated: we calculate each element in the output sequence by taking its corresponding input sequence element, and calculate its weights against all other elements in the input sequence as dot products, so that we end up with one weight for each element in the input sequence. Then we apply a softmax function to the weights, so that they sum up to 1, and we multiply each element in the input sequence by its corresponding weight, and sum everything up (so, the output element is also a dot product). In other words, for each output element, for each element in the input sequence, the weight (w) express how that element is related to all other elements in the input sequence, with “related” defined by the learning task, and the output element corresponding to that input element is a weighted sum over the whole input sequence (Bloem, 2019). This operation is performed in parallel via vectorization and matrix operations.

2.3 Neural Architectural patterns and strategies

An Architectural Pattern or Strategy differs from an Architecture being a broader concept rather than a single or specific Architecture. It may in fact encapsulate, refine or abstract one or more Architectures. For example, Autoencoders and GANs are Patterns rather than specific Architectures; more specifically, they are Patterns implementable with any type of Architecture. An Autoencoder has a sand-timer shape, with some compressed dimensional representation, also called bottleneck, in its middle hidden layer; this Pattern can be achieved with any Architecture. A GAN is composed by a Generator and a Discriminator, which can in turn be implemented using any kind of Neural Network Architecture.

Also, many Architectures can be *composed* together (at least two architectures, of the same type or of different types, are combined), *refined* (specialised with additional constraints, e.g. sparse and/or variational Autoencoder) or *nested* into one another (Pachet et al., 2019).

Patterns and Strategies are derived from specific practical implementations of Neural Networks, where nowadays more and more often single vanilla Architectures do not appear in the State-of-the-art, but they are rather articulated and refined into more complex compounds of Neural Networks. For example, the conditioning Strategy consists in altering (or specialising, since we can then sample from a particular ‘subset’ of the total distribution space) the learnt distribution depending on some externally imposed conditions.

2.4 Summary of Generative Neural Networks Architectures and Patterns for Audio Synthesis

Architecture	Pros	Cons
FF Feed Forward		- Deterministic (same input will always yield the same output) - No interactivity
CNN Convolutional Neural Network	- Parallelism (unlike RNN, a long input sequence can be processed quickly as a whole and not recurrently. Single forward pass speeds up training). - Training is faster than RNN - Can explicitly compute likelihood, with tractable density function - Faster to train than RNN	- Not well suited for generating indefinitely or infinitely long sequences - Conditioning control rate is not real-time because of parallel computation
RNN Recurrent Neural Network (also LSTM Long Short-Term Memories and GRU Gated Recurrent Units)	- Causal (the current term in the sequence is conditioned only on past events) - Arbitrary input/output length - Sequence-to-sequence (input sequence length different than output sequence length) possible with encoder-decoder architecture - Can explicitly and exactly compute likelihood - Conditioning control rate is	- Slow to train (RNN can easily become very deep networks, where the vanishing gradient problem is common) - Slow inference, because of sequential generation (for an N-long output, it needs N recursions) - The synthesised audio is only coherent up to 10/100ms for non-hierarchical Architectures

	real-time	
Transformers	- Unlike RNN, they don’t suffer from the vanishing gradient problem - Self attention allows parallel computation and unbounded long term memory dependencies	- For long sequences, like in Audio, require large amounts of memory - At the moment implementing real-time inference is hard

Pattern or Strategy	Pros	Cons
VAE (Variational Auto Encoder)	- Non-deterministic - Learns a compressed representation of the data-set q (z | x), which can be useful for other tasks and used in other networks - Non-deterministic - Interpolation between latent variables is possible, thanks to the distribution given to the embedded encodings (centred around the origin) - Can explicitly compute likelihood - Can compute inference queries	- Output quality is usually lower than GANs - Can explicitly compute likelihood, with intractable density function (which can not optimised directly, but we have rather to derive and optimise lower bound)
GANs (Generative Adversarial Networks)	- Fast inference - Comparing to VAE, finer control over global conditioning via its latent variables - Output samples quality is generally higher than VAE (but not within the Audio domain yet)	- Can implicitly compute likelihood - Jointly training of Generator and Discriminator networks is hard and unstable, since learning has to be balanced between the two - Can not compute inference queries
Conditioning (also known as *context* in speech applications)	- We can alter the learnt distribution depending on some externally imposed conditions, so that we sample from a conditional probability	

	-given a set of conditions- rather than from the data-set probability distribution itself (Huzaifah & Wyse, 2020) - Directly influence and control the synthesis with intuitive high-level labels	

3. Synthesis control strategies

Synthesisers of different kinds of sounds also demand different control parameters, as the same synthesis control parameters can not reasonably almost never be used for different types of sound synthesis, because of the substantial difference between the real-world objects or components that make the sounds in the first place. For example, it would not be meaningful to control a rolling ball sound with the same controls used to control a fire sound (e.g. amount of oxygen, material being burnt, amount of flames, ecc.).
Hence, even though defining and building a Sound synthesis model which works well for all kinds of texture sounds is an extremely fascinating topic, it is also true that every synthesiser Architecture has to be different in the sense that it has to synthesise different types of textures and consequently expose different control parameters. Along the line of building systems capable of generating specific sound models, also called factories, it is worth mentioning the Sound Model Factory by (Wyse et al., 2022).
My work differs from this approach in the sense that I am not working with a Model Factory, but I am rather specialising in only some types of texture sounds by building specific Synthesiser Architectures, each of which is trained with different types of sounds and has different synthesis parameter controls, which are physically-driven, hence highly dependent on the physical model of each specific synthesiser.
Since Generative NN models model a dataset distribution, the controls afforded by these models offer a way to *navigate* and *explore* that distribution. The control can be done more or less smoothly, and mainly 2 types of interaction exist; nominal values, that are categorical label as classes names (e.g. rain, car engine, wind sounds), and ordinal/continuous values, that are numerical values with smooth changes between them (e.g. car engine RPM, amount of wind, ecc.).

Finally, it is worth comparing this numerical controls-based approach with text and language-based user interfaces for sound generation, which are gaining much attention lately, like AudioLM (Borsos et al., 2022). Given the very different nature of these two kinds of control, and the nature of texture sounds, it is hard to imagine that text-based control interfaces can overperform the fine-graded nuances of smooth continuous numbers, for Sound Designers (with an exception for multimedia and multi-modal Sound Design, as automatic Video sonification is also a field of active research), and even more for Video Games, where the audio synthesis can be controlled in real-time by game parameters.

3.1 Nominal and numerical controls

My work focuses on numerical values as control variables, as they are the main means of control in classical Synthesisers. Class or event labels can not control, at least not finely enough, the nuances of the variables, quantities and entities involved in the physical generation of texture sounds (Okamoto & Imoto, 2020).
Sound Synthesis systems that use sound event labels (e.g. ‘rain’, ‘car engine’, ‘kick’, ‘snare’) as control parameters allow control only on the *type* of sound generated, with no control over the *gradations* of the high-level synthesis parameters that abstract the physical interaction of the sources (e.g. ‘amount of rain’, ‘car engine RPM’, ‘percussion boominess’).
In terms of Machine Learning implementation, this difference closely resembles the one between a classification task, where the output of the neural network is the class, or the classes which the input belongs to, and a regression task, where the output of the neural network is a continuous value representing a numerical prediction.
This fundamental choice of direction in control type reflects downstream over the entire implementation of the system and poses many central constraints, firstly on the dataset(s) used to train it.
Finally, for textures sounds, the high level controls do not always necessarily and directly affect the underlying sound at the sample level, but they rather do so in a statistical way, also called stationary parametrization (Wyse, 2022).

3.2 Meaningful, perceptually-relevant high-level controls in Generative Models

In supervised learning, the category which the present work falls into, the most effective way to control what a network generates is by curating the training data. A network will not be likely to be able to generate data from a very different distribution than the distribution of the training set. Therefore, the first step before training a model is to make sure that what aim to generate is well represented in the training dataset. In Generative Models, after training the model, new data can be generated by randomly sampling from the networks' latent space.
For better and deeper control over the generation, though, random sampling is not enough.

To identify mappings between latent representations and meaningful characteristics parameters, multiple techniques (e.g. dimensionality reduction like PCA, supervised approaches like annotating the parameters in generated data) have been researched; often, though, the features projected in the latent dimensions while training do not closely and uniquely represent features of the training set that are perceivably relevant to humans.
In order to explicitly enforce what the dimensions represent, two methodologies can be applied during the training process: latent space *regularisation* and *conditioning* (Ramires, 2023, p. 89 - 90). It is important to note that these techniques need to be applied while the latent space is created, that is, during the training phase, because, as we just saw, understanding what each latent representation represents after training (when the latent space is already created) is inconvenient.
In regularisation, the latent space is enforced to have the wanted characteristics by using ad-hoc loss functions. A recent interesting approach in this sense has been researched by (Esling et al., 2023), by improving the control affordances of RAVE, a high-audio-quality, Realtime Audio Variational autoEncoder (Esling et al., 2021), which does not expose control parameters meaningfully disentangled from one another. In fact, even though some high-level interaction is possible (e.g. one dimension is usually correlated to loudness, another dimension is mostly correlated to spectral centroid, ecc.), it is not always clear what unique and specific perceptual parameter is changed by a single latent dimension. This is reasonable and expected because RAVE is a universal model, designed to work with any type of sound rather than specific sounds

requiring specific synthesis control parameters. Specifically, since most of the 128 encoded latent dimensions end up representing noise, a distinction is made between informative and non-informative latent representations using Singular Value Decomposition -SVD-. Usually, 4 to 16 meaningful dimensions are kept from the 128 latent representations (Esling et al., 2021). It is worth noting that the perceptual nature of these controls, even though they are high-level and 'informative', heavily depends on the Dataset (more varying features in the data, tend to end up being represented in the latent dimensions), is potentially different at each new training performed on the Architecture, can not be known in advance, and do not represent arbitrary synthesis control parameters defined before training (what a particular dimension means to humans has to be *discovered* after training). This aspect, not present in the research aims, as we saw already, mainly because of its universality, is a substantial difference between (Esling et al., 2021) and the present Project.

In RAVE, the latent space regularisation is done with an adversarial criterion, and the generation is then conditioned with the given set of continuous controls. Specifically, the work done by (Esling et al., 2023) extends the Fader Networks (Lample & Zeghidour, 2018), by offering not binary-only (class-based) parameters, but continuous, high-level controls.

The experiments carried by (Esling et al., 2023) on continuous-valued descriptors, even though not performed on texture sounds but rather on instrumental, percussive and speech sounds, use, in addition to spectral descriptors, timbral features like *sharpness*, *boominess*, *hardness, depth, brightness, roughness, and warmth*, computed with the Audio Commons extractor (University of Surrey & AudioCommons project).

These features, also used by (Ramires, 2023) for percussive sound synthesis, are largely ascribable to what is pursued in this Thesis, being cognitively similar to physically-driven synthesis control parameters.

Conditioning, on the other hand, is based on re-thinking how the training process is carried out; extra information about our data is used as a conditioning signal c in order to train the model to learn a conditional density distribution p(x|c) instead of the distribution p(x) we were trying to learn previously. In a way, p(x|c) is a meaningful subset -as long as the conditionings are meaningfully representing what parameters we want to learn from the training data- of p(x). As we already saw, nominal types of conditionings do not carry much information, while numerical types of conditionings

allow the model to learn more subtle nuances from the data, which, during inference, can then be exposed to the user as control parameters (Ramires, 2023).

In Multi-tier Conditional RNN -MTCRNN- by (Huzaifah Bin Md Shahrin, 2020), even though high-level parameters are exposed, since the aim was to create a general model that can work with any texture sound and no focus has been put on learning specific meaningful controls for specific type of sounds, these controls -MFCCs (Huzaifah Bin Md Shahrin, 2020, p. 94)- are not based on intuitive synthesis characteristics nor physically-driven.
This is a fundamental difference between MTCRNN and the present work, otherwise very similar.

(Lundberg, 2020, p. 25), for 4-stroke car engine sounds synthesis, exposes a control representing the rises per minute -RPM- of the engine, as this value is expressed by the fundamental frequency f0 of a Harmonic plus Noise model, implemented with DDSP.

4. Generation of non-pitched sounds

The generation of harmonic pitched sounds (e.g. the ones that can be ascribed to musical instruments emitting a perceivably clear fundamental frequency, such as piano, guitar, ecc.) using deep learning has been and still is the object of an important quantity of research; the advent of Differentiable Digital Signal Processing or DDSP (Engel et al., 2020), combining traditional DSP with backpropagation and Machine Learning, paved the way for real-time timbre transfer synthesis techniques, where the timbre of an input sound is transformed into the timbre of a target sound, with possible interpolations between the two.
Mostly, though, the applications of this technique are restricted to DSP blocks which build up to the Harmonic plus Noise model (Serra & Smith, 1990), and are hence restricted to harmonic pitched sounds, inherently different from texture sounds, which are by their own nature harder to model, being much less deterministic and based on stochastic processes. Furthermore, DDSP only offers pitch and loudness controls over the generation, which, while being probably enough for musical instrument-based timbre transfer, it is certainly not appropriate for texture sound synthesis.

Overall, the synthesis of unpitched sounds, either percussive, environmental and textural, has been largely unanswered.

4.1 Drum sounds synthesis

Even though inherently different to texture sounds (but much more similar to them comparing to pitched harmonic sounds), drum sounds retain a level of similarity with textures mainly because of their variety (perceptually different at each new generation) and stochasticity (even different across new generations, they still have, with the same control synthesis parameters, the same statistical distribution).
On the other hand, the main difference between drum sounds and textures sounds is their time duration; short for the formers, indefinite, possibly infinite for the latters.

In non-Deep Learning Sound Design software, normally, an effective way of controlling drum sounds synthesis is to display parameters related to the synthesis process such as pitch, energy envelope or effects applied (Ramires, 2023). On the other hand, physically-driven controls, for Physical Modelling synthesisers, afford direct control over the physical properties of the sound sources or objects involved in the production of the sound. These objects can represent real-world elements, but not always and not necessarily; they can in fact simulate properties of percussive sound sources which would never have a chance to happen in the real world (e.g. oversized snare, ecc.).

As previously mentioned, (Ramires, 2023, pag. 96) used Wave-U-Net (Stoller et al., 2018) for synthesis of one-shot percussive sounds since long time dependencies are not needed for this type of sounds.

4.2 Textures sounds synthesis

(Farnell, 2010), before the Deep Learning era, coined the term *Procedural Audio*, referring to sound that is generated at runtime that typically utilises a synthesis engine driven by real-time parameters, based on a set of predefined behaviours which are attributed to the sound source -as well as all the objects involved in the sound generation- that is being emulated. Being loosely definable as a real-time, every-day sounds oriented (as opposed to musical instruments sounds emulation) subset of

Physical Modelling, Procedural Audio tries to solve the same problems, at least conceptually, as the work done with this Thesis.
The short-time benefits (real-time, automated, meaningfully controlled Sound Design, without using pre-recorded and looped audio files) are also comparable.

(Comunità & D. Reiss, 2021) used WaveGAN, a convolution-based network for unsupervised synthesis of raw-waveform audio, in order to generate footsteps sounds. While WaveGAN only was used as generator, WaveGAN and HiFi GAN (previously implemented for speech synthesis) were both tried as discriminators for 16 kHZ sounds. Even though perceptual tests yielded slightly different results in terms of perceived background noise, both Architectures provided high level of realism in the generated sounds; however, no focus has been placed on affording a priori-determined high level controls.

(Lundberg, 2020) used supervised learning and DDSP to reproduce 4-stroke car engine sounds. Even though the harmonic-plus-noise model of DDSP overall captures the most significant characteristics of engine sounds, artefacts in the generated sounds were especially noted in correspondence of transients, which occur at stochastic times in 4-stroke car engines. This aspect is most probably due to the deterministic nature of the DDSP Autoencoder model, which is not a Variational Autoencoder and hence tends to always generate the same output sound given the same input controls, which in the case of DDSP are of course not physically-based but something more musically-based like f0 (fundamental pitch) and amplitude.
We can thus formulate that DDSP, even though very appropriate for real-time applications, is not an appropriate method, given its harmonic-model-based nature, for neural texture sound synthesis.

5. Datasets

The scarcity of research in texture sound synthesis is both the cause and the consequence of the lack of big, well and numerically-annotated, synthesis oriented datasets for texture sounds. Since most of the work on Environmental sounds has been carried out for Acoustic Scene Classification, most of the datasets are oriented towards this objective and are thus annotated with nominal values -class names- rather

than numerical values -continuous values, e.g. controllable with a knob or a slider- representing sound synthesis parameters. In particular, the insufficiency of synthesis-oriented texture sounds datasets with physically-driven annotated parameters, is also due to the inherent complications that arise when trying to annotate statistically-based sounds with continuous values.
The first adversity is the time-resolution of the annotations, related to the duration of the sound to be annotated; is one annotation enough for each sound ? Are the synthesis parameters considerable static and statistically stationary for the entire duration of the sound ? Furthermore, the annotators should be expert, or at least have some solid knowledge, about the physics and Acoustics involved in the generation of the sound (they should be aware of e.g. what objects are implicated, how to they interact, what are the parameters of interest, ecc.).
As usual, it is worth making a comparison with the world of harmonic pitched sounds and related datasets for music Audio synthesis; for musical instruments sounds, the collection of audio clips can quite easily span the range of control dimensions like instrument, pitch, and velocity, with each clip tagged with the relevant combination of parameters. Creating an equivalent for texture sounds is difficult since factors of variation are not as clear-cut as compared to music, and replicating the entire range of variations in a controlled manner may prove to be a far-fetched endeavour.

There are very few datasets dedicated purely to audio textures. (Antognini et al., 2019) made the textures used for their work in CNN-based parameterisation freely available. However due to the nature of resynthesis, only one example per texture class is available; this is insufficient as training a deep generative model demands a much larger dataset with more variation.
Another possibility are environmental sounds datasets containing subsets of textures, including FSD50K (Fonseca et al., 2022), ESC-50 (J. Piczak, 2015), UrbanSound8K (Salamon & Jacoby, 2014), BBC Sound Effects Library (BBC, n.d.), and AudioSet (F. Gemmeke, et al., 2017). The main obstacle comes from the lack of detailed metadata, since the datasets are labelled with categorical variables rather than numerical variables, and clear and consistent factors of variation upon which control parameters can be built.

(Huzaifah Bin Md Shahrin, 2020, chapt. 6) engineered control parameters from audio features which were extracted directly from unlabeled audio data. While trying to model

and approximate any kind of texture sound with one unique system, this approach, since the extracted audio features, even if perceptually relevant, may be completely unrelated to any underlying physically-driven synthesis parameter, is far from guaranteeing a nice and smooth relationship with numerical controls and their acoustical causes and consequences.

5.1 Synthetic datasets

Even though they intrinsically do not sound exactly like real-world sounds, synthetically-generated datasets (generated by software synthesis, rather than recorded from a real-world sound source) have the fundamental advantage, over real-word texture sounds dataset, of being easily and precisely labelled. In fact, if the synthetic sounds were generated using Procedural Audio or Physical Modelling techniques, the numerical labels can just be the very synthesis control parameters used during synthesis.
At the cost of having less-real-sounding sounds, we have sounds which are finely labelled with numerical values, with a coherent 1-to-1 or 1-to-many mapping between those controls and the acoustical behaviours of the physical objects involved in sound generation.
Furthermore, other two important characteristics of synthetic datasets are the generation speed, the degree of automation in the generation process (the sounds are generated programmatically), and the high degree of customization due to the arbitrary variance given to the synthesis control parameters (which are automatically annotated in the dataset and will be used as ground truth during model training).

An important example of synthetic texture, annotated with physically informed parameters sounds dataset with continuous numerical annotations, is Synthetic Audio Textures, Syntex in short, by (Wyse, 2022), which is a collection of data set generators for texture sounds like chirps, pops, pistons, mosquitos, wind, wind chimes, peepers, bees, and applause. Models are downloaded as code which generates dataset locally with a single shell command.
Another synthetic dataset from the same author is Parameterized Soundset Database, PSoundSet in short (Wyse, n.d.), also used in the previously mentioned Multi-Tier Conditional RNN project by (Huzaifah Bin Md Shahrin, 2020).

The book ‘Designing Sound’ by (Farnell, 2010), is an important source for Procedural Audio real-time synthesisers, which could be quite easily produce annotated synthetic datasets with texture sounds like bouncing, rolling, creaking, fire, bubbles, running water, pouring, rain, electricity, thunder, wind, motors, fans, jet engines, footsteps, birds, mammals, and others. Similarly, the Sound Design Toolkit (*SOUND DESIGN TOOLKIT (SDT)*) is an open-source framework for ecologically founded sound synthesis and design.

For the sake of discussion, someone might argue the importance of training Neural Networks on synthetic sounds, which we already know the synthesis mechanisms of, since some software synthesis previously generated them. The obvious benefits are data-augmentation and transfer learning (prototype and pre-train a model before training it on real-world data).
To properly understand this last point it is important, then, momentarily focusing not on the aspect of learning whatever Sound synthesis procedure lies behind a particular sound, but rather on the aspect of learning what synthesis control parameters were used to generate a particular sound. Once we learned this, we can hopefully deduce the same mappings from real recordings (assuming that the synthetic and real sounds have some degree of common audio features; if not, the synthetic sounds may be post-processed in order to make them more similar to the real ones). The synthesis control parameters, built-in to some synthetic sound and extremely hard to annotate on real recordings, can in fact be used as ground truth in some parameter extractor neural network, that encodes the synthetic audio files into a few numerical synthesis control parameters. Once trained on synthetic sounds, the (frozen, not trained anymore) parameter extractor can be used to extract underlying synthesis control parameters from real texture sounds.

5.2 Data augmentation

Data augmentation, applicable to both synthetic and real datasets, is a technique used to reduce overfitting when training a machine learning model, by training models on several slightly-modified copies of existing data. New training examples, very similar to the original training dataset and with, for supervised learning, exactly the same annotations as the original training dataset examples, are generated; hence, both size and diversity of the training set are augmented. Parallely, since, mathematically

speaking, a bigger portion of the problem space is covered, model accuracy and robustness should also improve.
In the Audio domain, the transformations that are usually applied (to both raw waveform and spectrogram-based representations) are; time shifting, time stretching, noise addition, silence addition, impulse response convolution (e.g. reverb, ecc.), polarity inversion, filtering or frequency masking (e.g. high pass filtering, low pass filtering, ecc.), and random gain.

In the proof of concept of the present project, noise addition and gain randomization are used while creating the synthetic Audio dataset. This narrows the domain gap between synthetic and real datasets (naturally, there always is some noise into a recording of a real texture sound, and the waveform amplitudes of a real world dataset are never neither normalised nor recorded with exactly the same equipment and volume gain), and makes Domain Adaptation easier.

5.3 Transfer learning and synthetic-to-real Domain Adaptation

The present work will, starting from a labelled synthetic dataset and an unlabelled real dataset, take advantage of synthetic-to-real domain adaptation techniques.
When using a synthetic dataset to train a Neural Network model and later perform inference with that model over the real dataset, a major issue to take into account is how well, and how seamlessly, the trained model will transfer the learned mappings from the synthetic data to real data (usually, the model performance degrades substantially).
In other words, Domain Adaptation -DA-, a technique that seeks to align the statistical properties across domains (e.g., synthetic and real), so that Deep Learning models trained in one domain can be deployed in another, has paramount importance when dealing with synthetic datasets.
Such mismatch between the synthetic data -also called the ‘source’ domain- and real data -also called the ‘target’ domain- distributions needs to be regarded as domain differences, and various domain adaptation techniques have been proposed to align the distributions of synthetic and real audio samples in the feature space and alleviate the domain-gap problem.

Specifically, depending on the amount of annotations present in the real dataset, domain adaptation can be unsupervised (no labels present in the real dataset), and weakly/semi supervised (few labels in the real dataset). The former does not allow the possibility of fine-tuning the pre-trained model by performing a second training stage on the labelled real data, while the latter does.
Nevertheless, having a solution for an unsupervised domain adaptation task, gives automatically a solution for a semi-supervised domain adaptation task. Furthermore, in the context of unsupervised domain adaptation, the source domain is equivalent to the training data, and the target domain is analogous to the test data.

Quite obviously, the more realistic the synthetic data is -in other words, the more similar the synthetic data is to the real data-, the better the learned model will adapt to real data. Thus, an important consequence of this, is, inevitably, asking ourselves to what extent is it possible and expensive to build an extremely realistic synthetic dataset. In other words, bridging the gap between synthetic and real data is a very important aspect when creating classifiers for real data, trained on synthetic data.

5.3.1 Synthetic-to-real unsupervised Domain Adaptation with Domain Adversarial Networks

(Biswas et al., 2022), in the context of Sound Event Detection -SED-, try to minimise the difference between the source and target domains (also called domain shift, or distribution shift) by aligning their distributions. This is done by training an additional Neural Network to extract features present in both synthetic and real datasets, also called domain-invariant, or domain-independent features, since they are invariant to differences between the source and target domain.
Specifically, the Neural Network is a Domain Adversarial Network -DAN-, composed by three sub-networks; a feature generator, a label predictor / sound event detector (the example task for this particular sub-network is taken from the afore-mentioned paper, but any domain specific task, either classification or regression, can be performed) and a domain discriminator.
The feature generator is trained to produce domain-invariant features useful for the sound event detector to properly learn to perform its function, and are used as input for both the label predictor and the domain discriminator. The label predictor is trained to perform sound event detection on the generated features, but the task is generalisable; in other words, this sub-network learns to perform whatever task -supervised for the

source domain, unsupervised for the target domain- we want to adapt from the synthetic dataset to the real dataset; rather than taking as input two different distributions of two similar but not equal dataset, the label predictor takes as input lower-level representations of both datasets, so that the learned task will be applicable also to the real data with some deeper level of robustness. The learning of this sub-network is performed only on the supervised set of inputs; since ground truth data is available only for the synthetic data, backpropagation on the label predictor is performed only when the initial input data is synthetic, and not performed when it is real.
The domain discriminator is a binary classifier trained to distinguish the features passed by the feature generator between feature extracted from synthetic data or feature extracted from real data.
The feature generator (in this case, a Convolutional Recurrent Neural Network -CRNN-) needs to minimise the error of the Label Predictor (so that the model is good at discriminating) but also maximise the error of the Domain Classifier. If the Domain Classifier cannot distinguish between the features being produced when source and target domain inputs are fed to the Feature Extractor, then the features can be said to be domain invariant.

The overall intuition of Domain Adversarial Networks is to learn whatever task (classification, regression, ecc.) we want to adapt from synthetic to real data, in such a way that it can be performed with the highest degree of reliance possible over both labelled synthetic data and unlabelled real data. To achieve this, the representation of synthetic and real data is transformed into a non-intelligible series of number vectors, which are domain-independent.
From a statistical point of view, this means that we started with two different distributions (of the two datasets), and we ended up with embeddings (representations with lower number of dimensions, not perceptually meaningful or representative to humans) of both source and target domains in only one distribution.

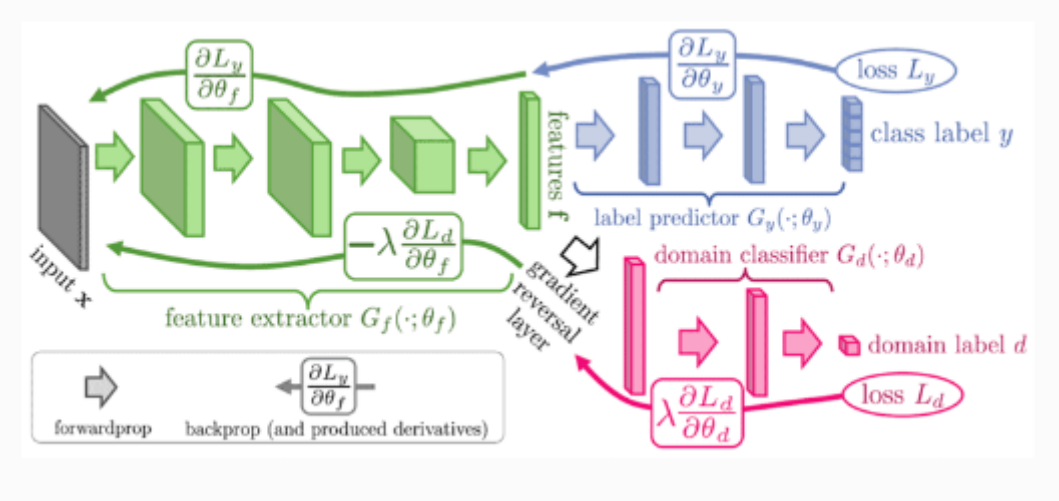


Figure 5.3.1 Domain Adversarial Neural Network architecture.

5.3.2 Evaluation of Unsupervised Domain Adaptation and hyper-parameter selection

To perform unsupervised domain adaptation, one should provide ways to set hyper-parameters (such as the domain regularisation parameter λ, the learning rate, the network architecture for our method) in an unsupervised way, i.e., without referring to labelled data in the target domain.
(Ganin et al., 2016, pag. 16) performed hyper-parameters validation (choosing the more correct hyper-parameter values) and early stopping (a form of regularisation used to avoid overfitting) with reverse validation. This work is a refining and specialisation of a previous traditional-DAN-based study (Ganin et al., 2015).

Given the labelled source sample S and the unlabeled target sample T, we split each set into training sets (St and Tt respectively, containing 90% of the original examples) and the validation sets (Sv and Tv respectively). We use the labelled set St and the unlabeled target set Tt to learn a classifier η, with which we label the previously unlabelled target dataset Tt (which becomes self-labelled). Now we swap source and target domain: the self-labelled dataset becomes our new source domain, and we remove the labels from St, which is going to be our new target domain. Then, using the same Domain Adversarial Network framework, we learn a reverse classifier ηr using the self-labelled set $\{(x, \eta(x))\}x \in Tt$ and the unlabeled part of St as target sample. Now, the reverse classifier ηr is evaluated on the validation set of the source domain (Sv), which is labelled.

We then say that the classifier η has a reverse validation risk of rSv (ηr).
The process is repeated with multiple values of hyper-parameters and the selected parameters are those corresponding to the classifier with the lowest reverse validation risk.
The validation set Sv is also used as an early stopping criterion during the learning of η, and self-labelled validation set {(x, η(x))}x∈Tv is used as an early stopping criterion during the learning of ηr. We also observed better accuracies when we initialised the learning of the reverse classifier ηr with the configuration learned by the network η.

5.3.3 Adversarial Discriminative Domain Adaptation

A generally simpler approach to unsupervised Domain Adaptation (to the context of Image classification) has been proposed by (Tzeng et al., 2017), with Adversarial Discriminative Domain Adaptation -ADDA-. In this work, a source representation mapping Ms is learned, along with a source classifier, Cs, and then these two networks are adapted to the target domain. This way, the source classification model, Cs, can be directly applied to the target representations, eliminating the need to learn a separate target classifier and instead setting, C = Cs = Ct.
The overall procedure is described as follows; first, pre-train a source encoder CNN using labelled source image examples. Next, perform adversarial adaptation by learning a target encoder CNN such that a discriminator that sees encoded source and target examples cannot reliably predict their domain label. Finally, during testing, target images are mapped with the target encoder to the shared feature space and classified by the same source classifier pre-trained on source domain images.
Generally, both source and target encoders (in the example of this very paper, CNNs) share the exact same Architecture (type of layers, number of layers, number of nodes, type of activation functions, ecc.), and usually the target encoder starts training with the same pre-trained parameters of the source encoder. However, recently, fully or partially untied weights (respectively, full and partially -with some parameters copied from the pre-trained source encoder- autonomous weights initialisation in the target encoder) have also been proposed.

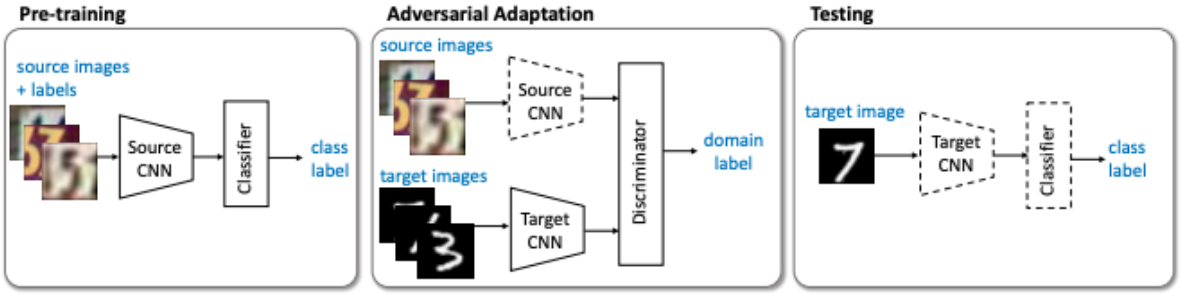


Figure 5.3.3. Adversarial Discriminative Domain Adaptation; dashed lines indicate frozen networks, with pre-trained parameters.

Comparing to (Biswas et al., 2022) and (Ganin et al., 2016, pag. 16) -which use Domain Adversarial Networks-, Adversarial Discriminative Domain Adaptation uses, as the name implies, only the Discriminator sub-network of a GAN framework, avoiding single-stage simultaneous training. Furthermore, ADDA implements two different feature representations encoders, one for each domain (only one encoder for both domains in DANs), but only one classifier/regressor for both domains (a classifier and a reverse classifier are implemented in DANs). Since there is no reverse classifier, once training and evaluation of the source classifier have been performed, there is no need in ADDA to re-evaluate or re-test the target classifier on the target domain.
In fact, since there is only one single classifier (the source classifier), the target encoder learns to encode the target domain into some domain-independent features, not distinguishable from the source encoding representations (learned in the Pre-Training stage); in DANs, conversely, is the classifier that learns to adapt to the domain-independent features, rather than forcing the target features to be as similar as possible to the source features.
Hence, training in ADDA is simpler than DANs because it is multi-stage rather than single-stage, and sequential rather than simultaneous where the loss of each sub-network depends on the loss of other sub-networks (each training stage is completely modular and independent from the others, rather than monolithic in one single stage).

As far as this Thesis is concerned, Adversarial Discriminative Domain Adaptation is used to create a synthesis control parameters extractor robust to work on real

-unlabelled- texture sounds. These very synthesis control parameters will be later used as conditioning inputs for the synthesis stage.

6. Summary of popular Generative Neural Networks implementations for Audio Synthesis

Architecture	Pros	Cons	Universal
WaveNet Dilated convolutions	- Clear caption of Audio in training data	- Long term characteristics are not captioned (unconditioned speech generation produces bubbling, as the time dependency does not last more than 300 ms, that is the last 2-3 phonemes)	Yes
SampleRNN Hierarchical RNN	- Clear caption of Audio in training data	- Long term characteristics are not captioned (HUZAIFAH, 2020, chapt. 2.5)	Yes
MTCRNN Hierarchical conditional RNN	- High level, even though not physically-derived controls	- The use of RNNs imply the adoption of a multi-tier hierarchical architecture in order to deal with longer time dependencies	Partially (texture sounds, controls are fixed high-level descriptors)
(Ramires, 2023) uses Wave-U-Net (Stoller, 2018) to generate one shot drum sounds from high-level timbral features descriptors	- Continuous, high level and meaningful spectral parameters controls	- Short, one shot sounds	Partially (one shot percussive sounds, controls are fixed timbral

			features)
RAVE (Esling et al., 2021) Real-time variational Auto-Encoder	- Works well with any type of sound - Real-time - Meaningful controls	- Perceptual nature of controls is not arbitrary and depends on many factors such as Training Dataset	Yes
DDSP Differentiable digital signal processing	- Traditional DSP knowledge usable in ML	- Only works well with pitched harmonic sounds	Yes

7. Practical implementation and Software framework

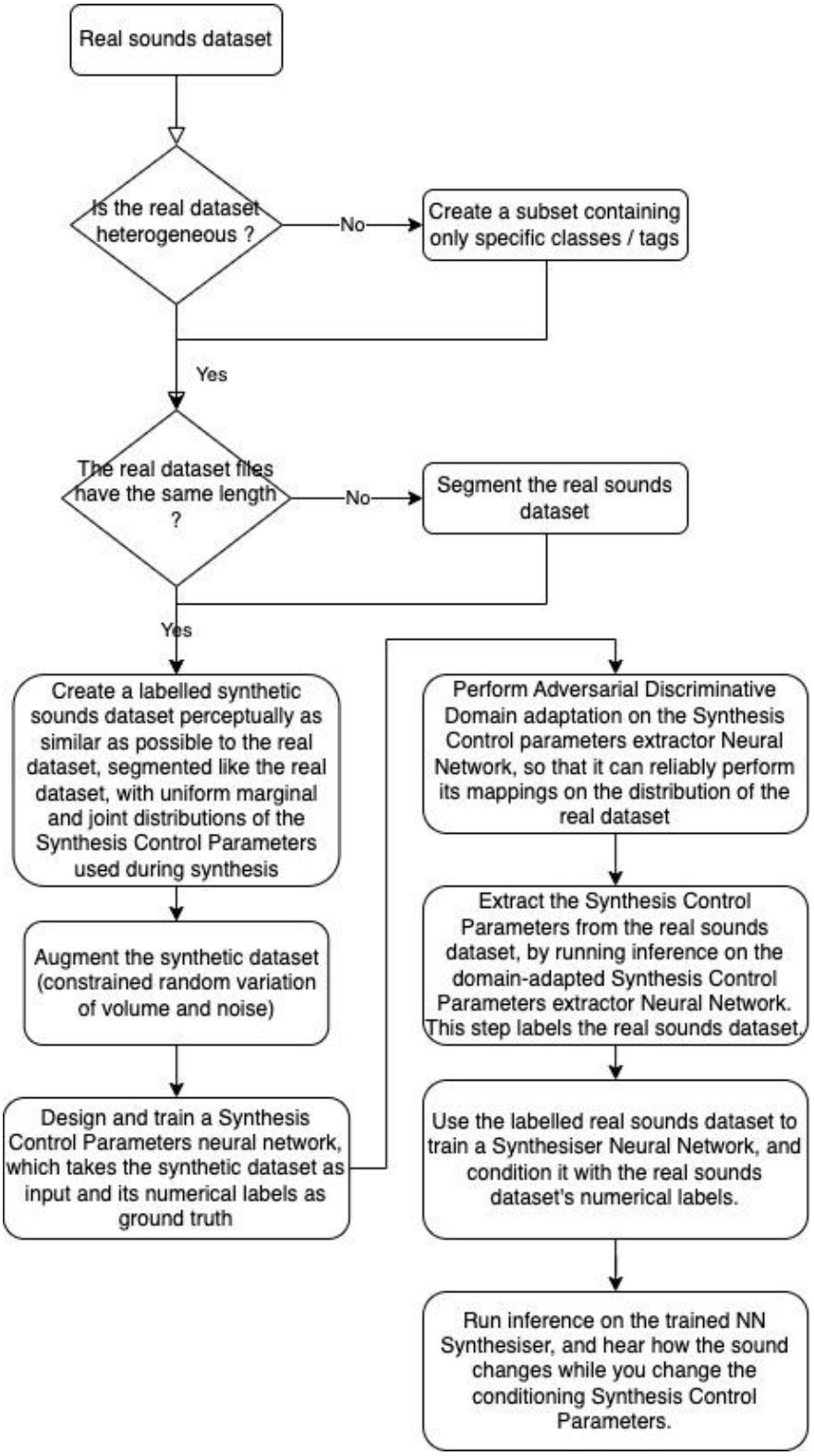


Figure 7 System overview - diagram.

7.1 Overview

The main objective of the proposed Neural Network pipeline is to generate, while exposing high-level, numerical and physically-driven control synthesis parameters, realistic texture sounds starting from unlabeled data sets of real texture sounds, and labelled datasets of synthetic texture sounds.
Each class of texture sounds intended to be synthesised has its own Deep Learning pipeline with unique and specific synthesis control parameters, composed mainly by two different and independent models; an Encoder and a Synthesiser.
For the creation of a proof of concept, and for the sake of the practical Project , waterflow texture sounds are used as an example, but the whole pipeline, both applied Research and practical project, are applicable to any kind of sound.
The prerequisites and starting points for using the framework are;

- Some kind of physical modelling / procedural audio-based Audio Synthesiser Engine for synthesising the synthetic version of the sounds we want the Neural Synthesiser to generate
    - Labelled synthetic dataset, augmented with noise addition and gain randomisation
- Some real-world sound dataset, to which the sounds synthesised by our Neural synthesiser need to sound the same
    - Unlabelled real dataset

First, it is worth explaining some specific nomenclature we are going to use. By ‘synthetic’ audio(s) (or synthetic sound) we intend audio files generated with software-based, Procedural Audio or Physical Modelling techniques; in other words, texture sounds which are not recordings of real-world texture sounds, but synthesised programmatically from scratch with a computer, and thus are expected not to sound exactly as a real sound, and as realistic as a recorded texture sound.
By ‘real’ sound we mean a recording of a real-world texture sound.
By ‘artificial’ we mean a sound generated by this Neural Network Architecture which is supposed to be as close as possible to a real sound; in other words, an artificial sound is a synthetic sound which can be misunderstood for a real world sound, hence sounds real, even though it’s artificial.

The pipeline is mainly divided into two parts, the first of which is preparatory for the second;

- Labelling the real world dataset with continuous, physically-driven variables
    - Create a synthetic dataset with a Physical Modelling Audio engine (based on traditional DSP, not Neural), using the Audio synthesis variables as labels
    - Supervisedly training of a Synthesis Control Parameters extractor Neural Network, which extracts the control variables from the Audio files of synthetic texture sounds
    - Adapt the Synthesis Control Parameters extractor from the source domain distribution (synthetic dataset) to the target domain distribution (real dataset)
- Use the aforementioned labelled real dataset to conditionally train a Neural Synthesiser, capable of producing artificial sounds according to some numerical control, representing a physical control dimension

The code base for creating the datasets and the Neural Networks, developed in Python and PyTorch, can be found on GitHub at https://github.com/Metiu-Metiu/Neural-Texture-Sound-Synthesis-with-physically-driven-continuous-controls.
Each of the subfolders contained in the repo contains README files explaining how to use the provided Software Framework, which has been made with the purpose of being extensible, reusable and scalable: the codebase is designed and structured in such a way so that it can be easily adapted to other kinds of sounds, to other kinds of Synthesis Control Parameters, to other datasets, and to other Convolutional-based Neural Networks Architectures.
Furthermore, each of the 4 sub-parts/folders (listed in the *'Project pipeline'* section) exposes an interface to the user, which mainly consists of:

- a JSON-like dictionary configuration file, which is used to specify the parameters of the intended task (e.g. the number of sound samples to create, the duration of each sound sample, the tags to take into account when extracting the sub-set of the canonical dataset, etc.), and it is designed to be as human-readable, portable, and human-friendly as possible.
- one or more Python scripts, runnable from the command line. The user does not need to modify the Python scripts; it is just necessary to modify the JSON files and run the relative Python scripts in the right order, as specified in each subfolder's README file.

The created Neural Networks and Datasets, useful for the use case at hand (waterflow sounds), can be respectively found on GitHub at https://github.com/Metiu-Metiu/Neural-Texture-Sound-synthesis---trained-Neural-Networks and https://github.com/Metiu-Metiu/Neural-Texture-Sound-synthesis---data-sets.

7.2 Project pipeline

7.2.1 Creation of synthetic Audio datasets

The codebase is at https://github.com/Metiu-Metiu/Neural-Texture-Sound-Synthesis-with-physically-driven-continuous-controls/tree/main/Creation_of_synthetic_Audio_datasets.

As explained in the chapter 'Datasets', synthetic sounds datasets provide us the fundamental benefit of having built-in continuously-annotated synthesis parameters, which are tedious, if not impossible, to achieve with real texture sounds dataset.
This codebase is dedicated to software designed to create Synthetic Audio Datasets with annotated synthesis control parameters, which can be used as ground truth in supervised Machine Learning applications.
Specifically, this project takes care of generating software that controls external 3rd parties Procedural Audio synthesis engines, without generating sounds itself; it rather focuses on managing probability distributions of synthesis control parameters values. 3rd parties synthesis control engines are only slightly changed in order to receive OSC messages from this project's python script. An example of external Audio synthesis engine, used in this Project as an example case, is the SDT_v2.2-078 (Sound Design Toolkit - https://github.com/SkAT-VG/SDT) Audio Engine, based on Max/MSP (an interesting feature of it is that, by setting the Audio driver as 'NonRealTime', very large synthetic datasets can be created off-line, as fast as a CPU can process them, without the need to wait for the entire duration of the Audio dataset itself).

7.2.1.1 Distribution of Synthesis Control Parameters in generated Synthetic datasets

Since the Synthesis Control Parameters network will be trained on the synthetic dataset, the variables distribution in the synthetic dataset is of paramount importance for the robustness of the Neural Network.

At the light of this, it is worth mentioning some basic notion of Statistics.
Given two random variables that are defined on the same probability space, the joint probability distribution is the corresponding probability distribution on all possible pairs of outputs. The joint distribution can just as well be considered for any given number of random variables. The joint distribution encodes the marginal distributions, i.e. the distributions of each of the individual random variables. It also encodes the conditional probability distributions, which deal with how the outputs of one random variable are distributed when given information on the outputs of the other random variable(s).

Even though, in the waterflow sounds example case provided in the codebase, only 1 Synthesis Control Parameter variable is used in the final Synthesiser, the synthetic dataset is nevertheless created taking account of 4 variables (the one used to control the final Synthesiser, and 3 other variables present in the Sound Design Toolkit/waterflow Max synth patch) in order to create a dataset capable of training a more robust parameters extractor and a more varied synthesiser (capable, with the same conditioning value for the only control used, of generating implicitly other novel values of latent variables).

7.2.1.2 Experiments on various distributions of Synthesis Control Parameters

Experiments have been carried out with the following distributions;

- Random uniform distribution
    - involves non-deterministic stochastic processes
    - yields a marginal uniform distribution, for each synthesis control parameter, as the number of audio files to be generated approaches infinity
- Uniform linearly spaced values distribution
    - marginal distributions are guaranteed to be uniform, but the joint distribution does not take into account all combinations of outputs between synth control param variables, hence it is not uniform

    - In order to generate synthesis control parameter values, no stochastic process is involved:
        - n linearly spaced values are generated (min and max values included) for each synth control param, with n = number of audio files to be generated
        - for each synth control param, there is 1 different value for each audio file and thus none of the values is repeated more than once across the entire dataset. The only stochastic process involved is the choice of which value to use for each audio file (in other words, a random choice is made for choosing the order of values with respect to the series of files).
- Uniform controllable variance linearly spaced values uniform distribution
    - both marginal and joint distributions are guaranteed to be uniform (the joint distribution is guaranteed to take into account all combinations of outputs between synth control param variables)
    - Priority is given to the user's choice of deciding arbitrary ratios between the number of unique values for each synth control param, useful when a synth control param needs to have different variance than others
    - Unique values are a set of non-repeated values for each synthesis control parameter, which will be repeated in the joint distribution as many times as needed so that every possible combinatorial match between variable outputs is covered. This is why the prompted number of audio files to be generated is not necessarily respected, since it is computed automatically and it is equal to the product of the number of unique values for each synth control param (which is computed automatically as well, by respecting the ratios between the number of unique values for each synthesis control parameter prompted by the user).

In order to train the Neural Networks in the most robust way, the chosen distribution for generating the synthetic dataset of waterflow sounds was the one affording uniform marginal and joint distributions (Uniform controllable variance linearly spaced values uniform distribution). For similar reasons, all the control variables were given the same variance.

7.2.1.3 Applications, results and challenges

A synthetic waterflow sounds dataset has been created (https://github.com/Metiu-Metiu/Neural-Texture-Sound-synthesis---data-sets/tree/main/SDT_FluidFlow_dataset_10000_1sec) with 10.000 sound files of 1 second duration each. Out of the sdt.fluidlow~ Max patch, 4 variables were considered;

- avgRate (10 different values)
    - Average n. of bubbles per second
- minRadius (10 different values)
    - Minimum bubble radius
- maxRadius (10 different values)
    - Maximum bubble radius
- expRadius (10 different values)
    - Bubble radius gamma factor

The total duration of the dataset is: 2 h : 46 m : 40 s.
Examples of the synthesised water flow sounds, for this specific example case, can be found at https://drive.google.com/drive/folders/1VVVnZHrPtPPod4mQeCIFIMBnFF5SIZst?usp=sharing.

Some synthetic audio files, or parts of them, happened to be completely silent (Neural Network usually should not be fed silent Audio files, as their performance will decrease, and they do not reflect real sounds scenarios). This problem has been fixed by carefully tweak synthesis control parameters ranges and volume of synthetic Audio files.

In early experiments, synthetic dataset with sounds of 3 seconds duration were created; these configurations were later discarded because the data fed into the Neural Networks would be unnecessarily too large.

7.2.2 Creation of real Audio datasets

Since all aforementioned real datasets include many classes of Environmental / Texture sounds, often very different between each other, and many pre-existing dataset

loader libraries do not offer comprehensive ‘filtering’ functionalities, a generic Software Framework has been implemented for this specific purpose.
In fact, this part of the Project is dedicated to software designed to take an already existing dataset of audio files and create a new dataset containing only a subset of the original dataset, with the possibility to segment the audio files into subsequent smaller audio files of a given duration (e.g. 10 seconds). Segmentation is needed in order to make all the durations equal, so that all files are easily loadable by Neural Networks, having the same input dimensions.
Chunks which would be smaller than the specified duration are discarded. Silent chunks (more often present, even in highest quality datasets) are also discarded, since they are to be considered as data out of the intended distribution for the Neural Network to be trained. The already existing dataset is called the 'canonical' version of the dataset.

Particular attention has been put on the criteria of labels/tags matching between the canonical dataset and the subset (the ‘keywords’ tag labels of the canonical dataset which need to also be present in the subset, or not present in the subset, in many combination and permutations). The user can in fact, in the JSON configuration file, specify 2 lists of string-type tags, one for the subset tags, one for the excluded tags (the latter can be empty).
For each file in the canonical dataset, the condition expressed by one of the following policies is checked; if satisfied, the canonical file is included in the subset, otherwise the canonical file is excluded from the subset.
The 5 options are:

- All and only the specified subset tags are present in the canonical dataset file
- At least all subset tags are present in canonical dataset file
- At least all subset tags are present in canonical dataset file and excluded tags are not
- At least one subset tag is present in canonical dataset file
- At least one subset tag is present in canonical dataset file and excluded tags are not

The script is designed to work with the soundata (Salamon & Fuentes) loader library and the FSD50K dataset, but the code is designed to be easily-extensible to other loader libraries and datasets as well.

7.2.2.1 Applications, results and challenges

In order to create a subset for our example case, based on water flow sounds, the tags extracted from the canonical dataset were 'Water' and 'Stream'. The avoided tags were all the tags related to water sounds, but not to water flow sounds; 'Rain', 'Gurgling', 'Steam' and 'Ocean'.
The policy used for the waterflow sounds example case is ‘At least all subset tags are present in the canonical dataset file and excluded tags are not’.

The resultant subset (https://github.com/Metiu-Metiu/Neural-Texture-Sound-synthesis---data-sets/tree/main/FSD50K_Water_Stream_subset_1sec) comprehends 4583, 1 second long audio files. Sample rate and duration of the produced Audio files were matched with the ones used for the synthetic dataset, as both synthetic and real datasets need to have the same input dimensions for any Neural Network trained on both datasets.
Since some silent Audio chunk was present, I needed to explicitly add the functionality to discard silent Audio segments while creating a dataset sub-set out of the FSD50K dataset. An ‘at least’ policy is used instead of an ‘all and only’ policy because otherwise the subset containing the ‘Water’ and ‘Stream’ tags would be very small.

Examples, at the light of the implemented example case, of real water flow sounds, can be found at: https://drive.google.com/drive/folders/1s-dY31UaUUhtDlge4wfN4pf0UaH3jE-s?usp=sharing.

7.2.3 Synthesis control parameters extraction from synthetic data and synthetic-to-real unsupervised Domain Adaptation

This module is the core of the present Master Thesis Project, as its aim is, to some extent, to make an unsupervised task (continuous-values labelling of the unlabelled real data) a supervised one (continuous-values labelling of labelled synthetic data).

Specifically, this module articulates in two main parts:

- Designing and training a Convolutional-based Neural Network, representing a Synthesis Control Parameters extractor, which is able to extract the Synthesis Control Parameters values from the audio files of the synthetic dataset

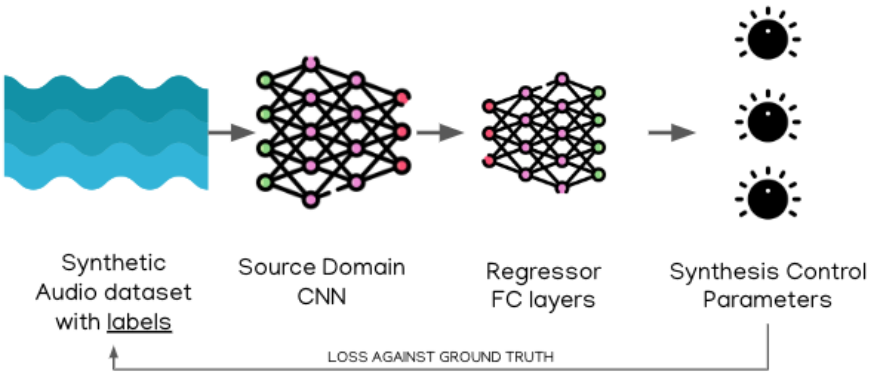


Figure 7.2.3.1.Synthesis Control Parameters extraction on synthetic data.

- Adapting the afore-mentioned pre-trained Neural Network to the distribution of the real audio files dataset, in order to be able to extract the Synthesis Control Parameters values from the real audio files of the real dataset. The general idea behind the implementation is to find cross-domain features by shifting the representation of the real data towards the distribution of the synthetic data, for which we already trained and evaluated a parameters extractor in a supervised way. Specifically, this module (inspired by the Adversarial Discriminative Domain Adaptation method presented in (Tzeng et al., 2017)) articulates in 3 main stages:
    - Design and train a binary sounds classifier Neural Network, able to distinguish between synthetic and real sounds, consisting on Fully-Connected layers only, which takes as input the (frozen, pre-trained and not learnable) last Convolutional Layer (in fact, the Flatten layer) of the afore-mentioned Synthesis Control Parameters

extractor network. This network is trained on the synthetic and real audio files datasets, in order to be able to distinguish between the two data distributions.

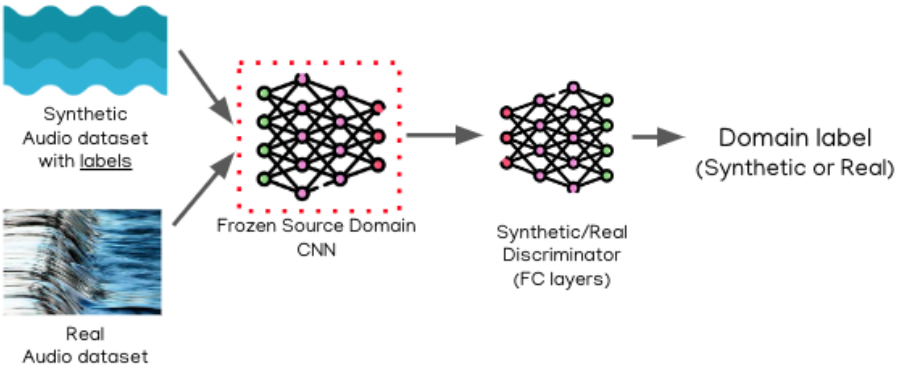


Figure 7.2.3.2. Domain adaptation - Domain classifier training.

- Design and train a new Neural Network, consisting of Convolutional Layers only, out of the pre-trained Convolutional Layers of the Synthesis Control Parameters extractor. This network is trained on the real audio files dataset, and its output is connected to the previously-created synthetic/real sounds classifier Neural Network. The aim of this network is to adapt the (target) domain distribution of the real dataset, to the (source) domain distribution of the synthetic dataset, in order to be able to adapt the Synthesis Control Parameters extractor to the real dataset, and to extract the Synthesis Control Parameters values from the real audio files of the real dataset.

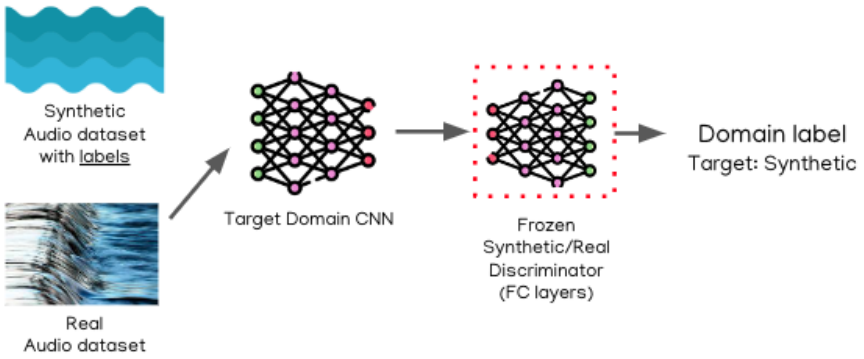


Figure 7.2.3.3. Domain adaptation - Convolutional layers shift from target domain (real dataset) to source domain (synthetic dataset).

- Extract the Synthesis Control Parameters from the real dataset by running inference on the frozen Neural Network, constituted by the domain-shifted Convolutional Layers (which, under the hood, extracts synthetic-domain relevant features from real data), and the original Fully Connected Layers of the Synthesis Control Parameters extractor Network, trained on the synthetic dataset.

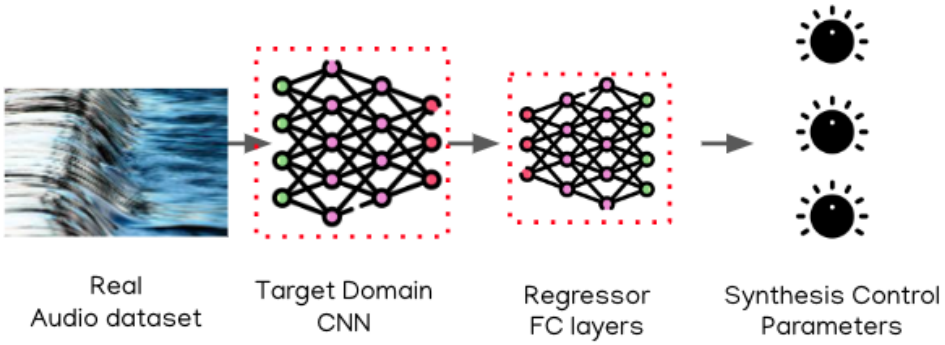


Figure 7.2.3.4. Synthesis control parameters extraction - inference on target domain.

7.2.3.1 Neural Networks design

The proposed framework includes Software functionalities which allow to create Convolutional-based Neural Networks (one or more convolutional layers, followed by a flatten layer, followed by one or more fully connected layers), at the light of the aforementioned parameters extraction and Domain Adaptation modules, in a dynamic

way by only providing high-level hyperparameters. The internal structure of the Networks adapt automatically to the requested shape of input data and to other configuration parameters. For example, the size of the Fully Connected layers' input is automatically calculated, so that the construction of the network architecture is completely automatic and dynamic.

In the first Convolutional layer, there is always a 1-to-1 mapping between a Synthesis Control Parameter and a Convolutional channel (in our example case, 4 Convolutional channels for 4 Synthesis Control Parameters). For each consecutive Convolutional Layer, the number of channels can stay the same or be multiplied by any integer number (in my example case, there is a multiplication factor of 2).
The same, but reversed, policy is applied in the Fully Connected layers with a divider rather than a multiplier (in my example case, each Fully Connected layer is ¼ the size of the previous Fully Connected layer).
This level of dynamic Architecture design allows fast prototyping and experimenting with various settings and hyperparameters.

The only a-priori decision is that Convolutional Layers are followed by the following Layers types as they proved to be effective:
- batch normalisation layer
- leaky reLU activation function
- Max Pool layer

The Neural Network designed for the proposed example case is lightweight (1 MB, see Figure 7.2.3.2 for further details), and all the training has been performed on a common CPU 2,9 GHz Dual-Core Intel Core i7.

7.2.3.2 Applications, results and challenges

Since the task is regression-based and the output values are normalised between 0 and 1, a leaky ReLU activation function proved itself to be the best solution, as ReLU functions would clip any negative value to 0, and the Network would not learn.

Data augmentation was applied to Synthetic Audio files by adding soft background noise. Specifically, some white noise was created and then filtered with a low pass filter, the cutoff frequency of which was randomly chosen between given ranges (50 hZ

- 300 hZ). The noise volume was also randomised between given ranges. This helped in making the synthetic dataset distribution more similar to the real dataset distribution even before the Domain Adaptation stage, while also improving the Domain Adaptation performance.

Pre-processing the input Audio files without waveform amplitude normalisation leads to modal collapse, a scenario where the Neural Network produces a limited set of outputs (very similar to each other), regardless of the incoming input.
Audio waveform normalisation (for the time being, each Audio file's waveform is normalised between 0 and 1/-1 with respect to itself, so that each Audio sample is normalised independently) helps in stabilising the distribution across the entire dataset. Without normalisation, each waveform had very different amplitude ranges compared to other waveforms.
Before any other transform, applying normalisation to the waveforms' amplitude is of paramount importance in order not to have modal collapse in the Neural Networks' output. Modal collapse is an unwanted situation where the Neural Network only learns to produce the same values, or almost the same values, regardless of the input. Since the waveform amplitude varies much across different samples, there is the need to make this variable uniform across the entire dataset (both synthetic and real) to train the Network in a reliable way.

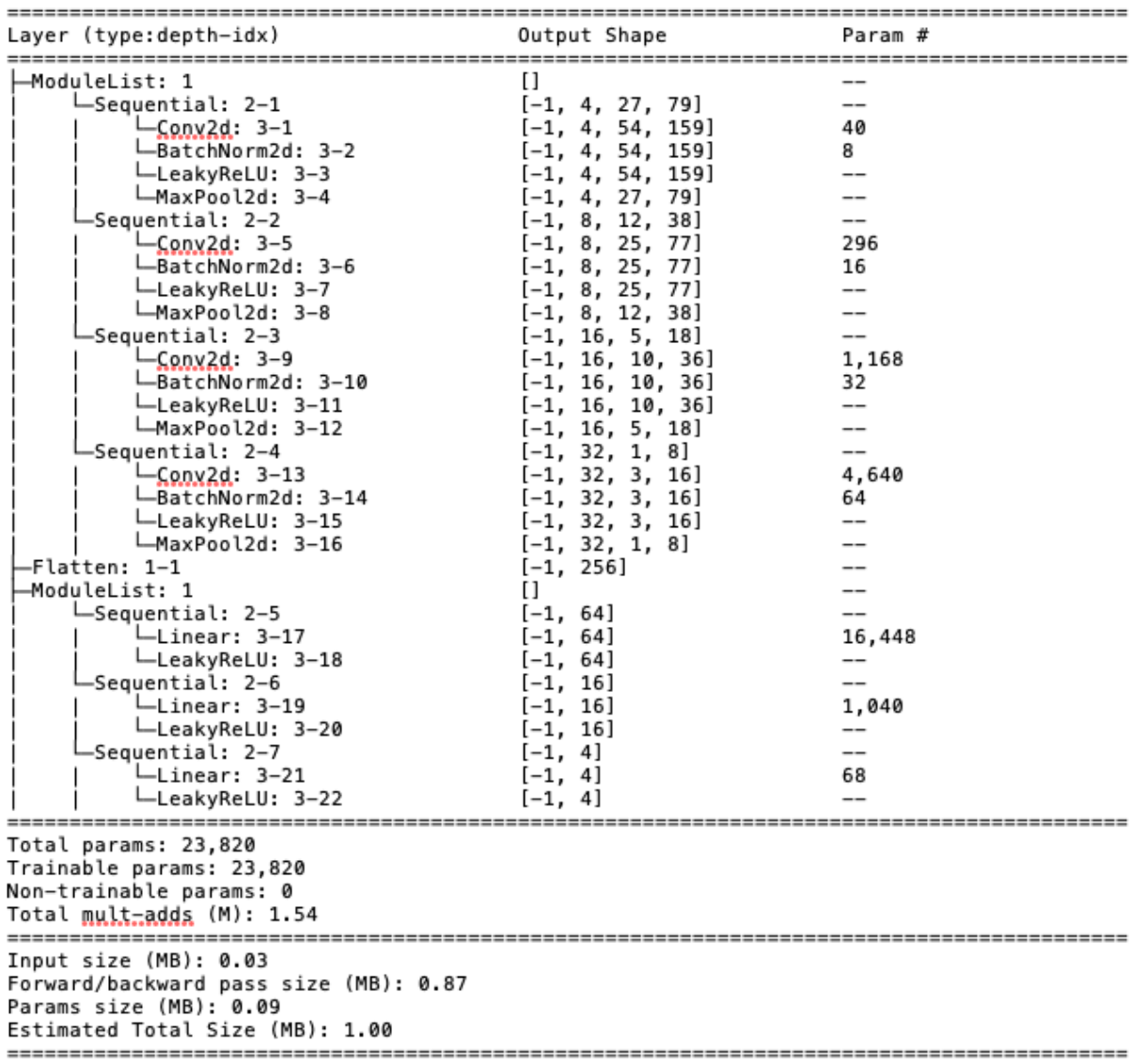
MelSpectrogram transforms, in the example case waterflow sounds, have been applied to the input sound files; a DB-scale conversion is also performed afterwards and the resulting tensor is normalised (this process yields much more robust networks).
The power representation out of the MelSpectrogram is normalised between 0. - 1., then converted into DB scale (ranging from -80 dB to 0 dB) and again normalised between 0. and 1.. DB scaling and normalisation improved model performance and, together with Domain Adaptation, helped obtain consistent ranges when performing Synthesis Control Parameters extraction over the Real Audio dataset.
While normalisation stabilised the distribution of the Spectrograms of the whole dataset, dB scaling and MelSpectrograms represent the Audios in a more humanly perceptually-relevant way.

As a general trend, 2D Convolutions, with MelSpectrograms as input, work better than 'raw' 1D Convolutions.

With smaller datasets it’s much easier to overfit, and the effects of this are seen at the early stages of training.

Dropouts did not increase performance.

Batch normalisation should be used after the Convolutional Layers, before the activation function, and it improved model performance.



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ModuleList: 1                          []                        --
│    └─Sequential: 2-1                   [-1, 4, 27, 79]           --
│    │    └─Conv2d: 3-1                  [-1, 4, 54, 159]          40
│    │    └─BatchNorm2d: 3-2             [-1, 4, 54, 159]          8
│    │    └─LeakyReLU: 3-3               [-1, 4, 54, 159]          --
│    │    └─MaxPool2d: 3-4               [-1, 4, 27, 79]           --
│    └─Sequential: 2-2                   [-1, 8, 12, 38]           --
│    │    └─Conv2d: 3-5                  [-1, 8, 25, 77]           296
│    │    └─BatchNorm2d: 3-6             [-1, 8, 25, 77]           16
│    │    └─LeakyReLU: 3-7               [-1, 8, 25, 77]           --
│    │    └─MaxPool2d: 3-8               [-1, 8, 12, 38]           --
│    └─Sequential: 2-3                   [-1, 16, 5, 18]           --
│    │    └─Conv2d: 3-9                  [-1, 16, 10, 36]          1,168
│    │    └─BatchNorm2d: 3-10            [-1, 16, 10, 36]          32
│    │    └─LeakyReLU: 3-11              [-1, 16, 10, 36]          --
│    │    └─MaxPool2d: 3-12              [-1, 16, 5, 18]           --
│    └─Sequential: 2-4                   [-1, 32, 1, 8]            --
│    │    └─Conv2d: 3-13                 [-1, 32, 3, 16]           4,640
│    │    └─BatchNorm2d: 3-14            [-1, 32, 3, 16]           64
│    │    └─LeakyReLU: 3-15              [-1, 32, 3, 16]           --
│    │    └─MaxPool2d: 3-16              [-1, 32, 1, 8]            --
├─Flatten: 1-1                           [-1, 256]                 --
├─ModuleList: 1                          []                        --
│    └─Sequential: 2-5                   [-1, 64]                  --
│    │    └─Linear: 3-17                 [-1, 64]                  16,448
│    │    └─LeakyReLU: 3-18              [-1, 64]                  --
│    └─Sequential: 2-6                   [-1, 16]                  --
│    │    └─Linear: 3-19                 [-1, 16]                  1,040
│    │    └─LeakyReLU: 3-20              [-1, 16]                  --
│    └─Sequential: 2-7                   [-1, 4]                   --
│    │    └─Linear: 3-21                 [-1, 4]                   68
│    │    └─LeakyReLU: 3-22              [-1, 4]                   --
==========================================================================================
Total params: 23,820
Trainable params: 23,820
Non-trainable params: 0
Total mult-adds (M): 1.54
==========================================================================================
Input size (MB): 0.03
Forward/backward pass size (MB): 0.87
Params size (MB): 0.09
Estimated Total Size (MB): 1.00
==========================================================================================
```

Figure 7.2.3.2. Synthesis control parameters extraction - PyTorch model summary for the implemented example case (extraction of 4 Synthesis Control Parameters). The input dimensions represent the Audio spectrogram representation of the Audio file.

7.2.3.3 Evaluation

To evaluate the effectiveness of the Domain Adaptation, both Domain-adapted and non Domain-adapted Synthesis Control Parameters extractor networks (trained on the same datasets in the same ways) were inference-run with real data as input and the produced continuous-valued labels were observed.

With no Domain Adaptation we can see that we obtain inconsistent results; values are sometimes greatly out of range (< 0. And > 1.) and modal collapse is an issue (though,

not as bad as when not applying waveform amplitude normalisation). Also, mean and standard deviation values deviate more from the ideal values (being 0.5 as mean and 0.5 as std deviation).
With Domain Adaptation, on the contrary, we can see that the values, as well as their mean and std deviation are closer to ideal results.

Non-normalised data	
Without Domain Adaptation	With Domain Adaptation
Minimum values: avgRate 0.152330 minRadius 0.105083 maxRadius 0.511343 expRadius 0.571902 Maximum values: avgRate 1.211358 minRadius 0.675838 maxRadius 0.829647 expRadius 1.775583 Mean values: avgRate 0.554509 minRadius 0.367322 maxRadius 0.632236 expRadius 1.144130 Standard deviation values: avgRate 0.088792 minRadius 0.057460 maxRadius 0.034437 expRadius 0.117516	Minimum values: avgRate -0.198639 minRadius -0.070783 maxRadius 0.450357 expRadius -0.577973 Maximum values: avgRate 0.691982 minRadius 0.880355 maxRadius 1.082828 expRadius 0.921750 Mean values: avgRate 0.437396 minRadius 0.276601 maxRadius 0.594658 expRadius 0.327418 Standard deviation values: avgRate 0.114125 minRadius 0.100925 maxRadius 0.084456 expRadius 0.166482

Figure 7.2.3.3.1. Synthesis control parameters extraction from real sounds dataset - non-normalised data analysis.

Data normalised between 0 and 1	
Without Domain Adaptation	With Domain Adaptation
Minimum values: avgRate 0.0 minRadius 0.0 maxRadius 0.0 expRadius 0.0 Maximum values: avgRate 1.0 minRadius 1.0 maxRadius 1.0 expRadius 1.0 Mean values: avgRate 0.379763 minRadius 0.459460 maxRadius 0.379804 expRadius 0.475398 Standard deviation values: avgRate 0.083843 minRadius 0.100673 maxRadius 0.108190 expRadius 0.097631	Minimum values: avgRate 0.0 minRadius 0.0 maxRadius 0.0 expRadius 0.0 Maximum values: avgRate 1.0 minRadius 1.0 maxRadius 1.0 expRadius 1.0 Mean values: avgRate 0.714148 minRadius 0.365230 maxRadius 0.228154 expRadius 0.603706 Standard deviation values: avgRate 0.128141 minRadius 0.106110 maxRadius 0.133533 expRadius 0.111008

Figure 7.2.3.3.2. Synthesis control parameters extraction from real sounds dataset - normalised data analysis.

To evaluate the perceptual quality and reliability of the Synthesis Control Parameters extraction procedure, 5 samples were randomly selected from the labelled real data set and the same Audio Engine used for synthesising the synthetic data set (in our

example case, Max and the Sound Design Toolkit) was used to synthesise back synthetic sounds using the same numerical values as the labels of the 5 real audio files (https://drive.google.com/drive/folders/1A7lUOoSTcEMUOXg6InfCkd14LyeQhVpG?usp=sharing). Then, a perceptual comparative test was performed and evaluated as sufficiently good.

7.2.4 Conditioned Neural Audio synthesis

Finally, the obtained labelled real dataset is used to conditionally train a Neural Network synthesiser. In order to perform good, unbiased training on the Neural Network synthesiser,
what is needed is an even distribution of data across the parameter values. In order to achieve this, a subset of the labelled real dataset is created with fewer samples and uniform distribution with respect to the Synthesis Control Parameter variable which naturally has a distribution as close as possible to a uniform distribution.

In the provided example case, an ad-hoc dataset has been recorded, and the 1-tier version of MTCRNN has been used. The dataset consist in 20 minutes recordings of a household shower water flow sound, recorded while manually adjusting the water flow amount (approximately every 2 minutes, the shower handle was turned clockwise 1/10 of its full range), and the model is implemented via Gated Recurrent Units (GRUs).

For preparing the data for the MTCRNN synthesiser training, 2 steps are involved:

- First, a numerical analysis of the labels of the real dataset is performed; min/max, mean and std deviation values are calculated for each Synthesis Control parameters. Then, a subset of the labelled real dataset is created separately, taking care of only selecting sound samples whose variables values distribution adds up to a uniform distribution. The purpose of this procedure is to train the synthesiser by weighting each 'class' of numerical controls equally, so that the synthesis stage is not biassed towards some values. Of course, we should try to retain as many samples as possible in the subset with uniform distribution. Specifically, an histogram is created out of the distribution of one variable, and the same number of audio files is selected out of each histogram bin; if not enough samples are available in a particular bin, the bin is discarded from the subset; if more samples are present in a particular bin, only n samples

are selected, where n is the selected number (to be as close as possible to the median number of samples for each bin).
In the presented example case, the variable which more closely resembles a uniform distribution is ‘expRadius’. Since almost all of the histogram bins contain at least 77 files, then we can have a uniform distribution of data across the parameter values if we take 77 files from each of the bins.

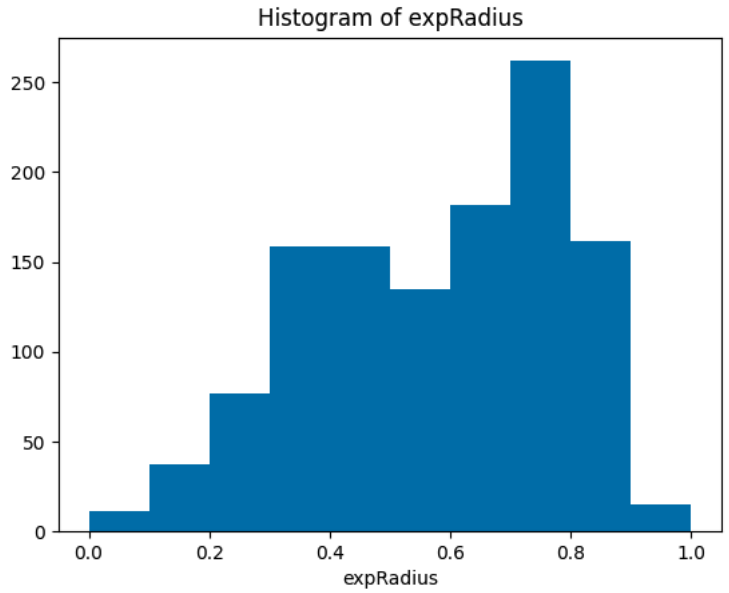


Figure 7.2.4. 10 bins histogram representing the distribution of the ‘expRadius’ Synthesis Control Parameter in the recorded shower dataset.

- Second, the real data set is resampled to a target sample rate (default = 16 kHz) and .params files are created for each audio file and relative numeric label. In fact, the MTCRNN model (multi-tier conditional recurrent neural network) by Lonce Wyse (https://github.com/Metiu-Metiu/MTCRNN) accepts this kind of file as conditioning input. The format of the .params file is just a text file containing JSON-like dictionary like the following;
  { "meta": { "filename": "RegularPopRandomPitch_r02.00.wav" },
  "parameter_name": { "times": [0, 10], "values": [0.0, 0.0], "units": "norm", "nvals": 11, "minval": 0, "maxval": 1, }, }

7.2.4.1 Applications, results and challenges

The MTCRNN model (Huzaifah Bin Md Shahrin, 2020) offers the possibility of multi-tier conditioning controls over the generation process, where, in cascade, higher-level controls, pertaining to longer temporal spans, control lower level controls, related to shorter temporal spans. Nevertheless, the practical use of the MTCRNN model for the example case implemented in this Project, considers only 1 tier/level of control, that is, the Synthesis Control Parameter dimension.

A 10 seconds Audio file generated by the MTCRNN model trained on water flow shower sounds and conditioned on the 'expRadius' variable, can be found at https://drive.google.com/drive/folders/1kuK28htgfQYP5uDLR3A8EHfOY0FlmLC_?usp=sharing. The 'expRadius' control variable increases linearly from 0. to 1. Throughout the entire duration of the Audio file.

7.2.4.2 Evaluation

The Audio file generated by the MTCRNN model is extremely realistic, but the perceived controlled variable is not perceptually and perfectly linear to the one used as numerical control during inference. In fact, a consistent part of the perceived variance sounds to be concentrated around the middle of the Audio file (between 4 and 5 secs), while the rest (beginning and end of the Audio file), yields less perceived changes in the ‘expRadius’ variable. In other words, there is a discrepancy between the dimension of the numerical control variable afforded by the System, and its dimension as perceived by perceptually relevant changes in the sound.
The two dimensions are not exactly and entirely linear to one another.

8. Conclusions and next steps

8.1 Summary

While researching the topic of Neural Sound Synthesis controllable with numerical values, a theoretical background, a state-of-the-art, a practical example case based on water flow sounds, and a documented, reusable and scalable Software framework have been created. Specifically, the practical Project involves;

- Creation of a (subset of a) real sounds dataset
- Creation of a synthetic dataset
- Labelling of the real sounds dataset
- Sound synthesis -producing realistically-sounding sounds- with numerical, physically-driven controls

Even if the end-goal of the Project is to synthesise some sound, a prominent part of the effort has been put into the systematic numerical labelling of some real sounds dataset, which is an essential and crucial step for the conditioned training of a Neural Network. In fact, probably the most important part of the present Project are the first 3 aforementioned modules, as the synthesiser Neural Network model is something re-used from the State-of-the-art.
Specifically, the Synthesis Control Parameters extractor Neural Network, is a fully customisable and dynamic Software class (https://github.com/Metiu-Metiu/Neural-Texture-Sound-Synthesis-with-physically-driven-continuous-controls/blob/main/Synthetic_to_real_unsupervised_Domain_Adaptation/Neural_Networks.py) for implementing custom CNN aimed to extract continuous-valued features from an Audio dataset.

Since the all 4 aforementioned modules are fundamental for addressing totally the Research goal behind the Project, the tasks have been implemented taking always into consideration the overall result, without focusing too much on any of them. This, of course, is not to say that there is no room for improvement.
In other words, while creating an example case (water flow sounds), it was important to pave the way for a re-usable and generalisable Software framework, and this latter aspect ended up needing more time than the former.

8.2 Contributions

The offered Python Software framework, being public, abstract, generalisable to any sound and to any numerical Synthesis control, will hopefully make Research and prototyping in numerically-controlled Neural Sound Synthesis easier and quicker.

For Generative AI and numerically-controlled Neural Sound Synthesis most of the performance-crucial part of the job is before the (last) Synthesis stage; extremely important are the data analysis, analytics and processing done on real and synthetic datasets.
This, even though there is room for improvement, is the focus of the current project, especially in terms of controlling the distributions of variables while synthesising the synthetic dataset.

To be really Generative, a System should produce different (or sufficiently varied) outcomes with the same input; in order to do so, the Synthesis Control Parameters extractor needed to map the same value of each variable to multiple values of all other variables. In other words, in order for the synthesiser to be truly generative and produce slightly different results with the same input, the Synthesis Control Parameters extractor should be as robust as possible to detect the whole space of conditional distributions in the variables taken into account.
In fact, it is good to have sampled many more dimensions in the parameters extractor (4, in the provided example case) than the number of variables actually used in conditioning and controlling the final synthesiser (1, in the provided example case), because the robustness of the final synthesiser, since its conditioning variables are created with the Synthesis Control Parameters extractor, depends deeply on the robustness of the Synthesis Control Parameters extractor. The more variables we consider in the extractor, and more frequently we sample them, the more robust the whole system is. This, of course, applies to the 4 variables of the proposed example case, but necessarily applies to any number and quality of variables at hand.
The implemented Software framework is completely transparent with respect to the number and quality of variables, that is, it is usable with any number and type of numerical variables.
A possible future way of Research would be, for example, to keep a very high number of numerical samples in the variable which will be the final control of the synthesiser,

and sample the other variables much less often. This approach would be a compromise between distribution span, system performance, and dataset size.

8.3 Limitations and future work

A more robust and systematic method of evaluation should be researched and implemented for analysing and evaluating the -numerical, not only perceptual- correlation between the real sounds dataset, and the synthetic sounds dataset, both before and after the creation of the synthetic dataset.
In fact, the synthetic dataset should resemble as much as possible the distribution of the data variables in the real dataset; since we are not interested in Audio level representations (MFCC, spectral centroid, ecc.), how can we control the synthetic dataset generation process so that the synthetic data is as much as possible similar to the real data, is still an open question.
The systematic and comprehensive implementation of such a system was left out of the present Project mainly because of time constraints, but the developed Software framework allows now to focus on it and use the rest of the System to research numerical correlations policies and evaluate them, by seeing in cascade its effect on the final synthesised sounds.
In fact, the issue of dimensional correlation between real and synthetic data, reflects downstream from the very first stage (creation of a synthetic dataset / subset of a real dataset) to the other successive submodules of the pipeline; the Audio file generated by the MTCRNN model is extremely realistic, but the perceived controlled variable is not perfectly linear to the one used as numerical control during inference.

Furthermore, in the provided example case, one interesting thing is that maxRadius is the parameter that worked best. It is the parameter that described the range of sounds in the natural dataset best, while ‘avgRate’ was expected to work best since it was the most perceptually important physical control property in the Physical Modelling Engine used for creating the synthetic sounds dataset. Again, correlating these dimensions to their effective counterparts in the real sounds dataset, is probably the one and only real improvement, nevertheless crucial, that the implemented System really needs.

A possible future improvement, in this sense, about the specific provided example case, would be to re-run the whole pipeline by creating a synthetic dataset with more densely-sampled variables, even though it would of course create a much bigger

synthetic dataset (e.g. 20 values per variable, 20 * 20 * 20 * 20 = 160.000 sound samples rather than 10.000 sound samples). By doing so, even without a strongly motivated correlation between real and synthetic datasets, the broader distribution of numerical variables would ensure a better approximation of both synthetic and real datasets distributions.

Also, other types of sounds could be investigated to better explore and validate the relationships between the synthesis control variables of the real dataset, to be reproduced as closely as possible in the synthetic dataset.
Since creating a Physical Modelling Audio Synthesiser from scratch in order to generate more custom synthetic dataset would probably have been the entire scope of a separate Thesis Project, and there was no time for that, it has been decided to re-use an existing 3rd party one. This, of course, limited the horizon of possible real sounds compatible with the synthetic dataset afforded by the 3rd party DSP synthesiser.
Now that there is a functioning Python framework, future Research could focus on improving particular aspects related to specific example cases.

RNN/GRU, used under the hood by the MTCRNN model, are for the moment pretty far from real time in terms of Sound Synthesis inference; they synthesise sounds around 10 times slower than real time. Possibly, conditional CNN-based models or Transformer-based models could be used for the final Synthesis stage.

8.4 Acknowledgments

Very little of this Project would have been possible without the patience, guidance, and help of my Thesis Supervisor Lonce Wyse. I sincerely think I could not choose a better supervisor, not only for the choice of the Research topic and his experience with Generative AI, but also for his dedication, kindness and availability.

Secondly, I would like to thank my girlfriend Giulia for the immense, amazing support during my last year in Barcelona. Even though far away, she gave me so much strength against stress and hard times.

Then, I would like to thank my old good friend Andrea, also for the support, the code reviews, the long phone calls, and for being with me during my Thesis presentation in July.

Lastly, I would like to thank all the Faculty members of the Sound and Music Computing Master, as well as all my colleagues, Dilip in particular, for having taught me so much during the last year, both technically and humanly.

It has been a tough year, but it has definitely been worth it and I guess I am already starting to see the fruits of it !