Audio Sample Generator

Richard Meyer

Dec 2023

# Description

The overall goal is to be able to implement a synthesizer that can generate new sounds in an interesting manner without the user having to tweak 100s of parameters.

The synthesizer itself would be written as an audio plug-in, typically written in C++ using libraries such as JUCE and released as AU and VST.

Machine Learning hopefully provides an opportunity to create interesting new wave forms. The google NSynth released in 2016 aimed to do something similar, this leverages the WaveNet research.

My proposal here is to create a deep-learning model that can regenerate audio samples of musical instruments reasonably faithfully. The model should then be able to generate new samples, either by interpolation between samples, or randomly. The "reasonably faithfully" is important, the goal is not to replicate existing sample libraries, but rather to generate new and interesting sounds.

# Data

The model is trained on a dataset of public domain samples acquired from websites such as https://freewavesamples.com.

Currently, over 1000 samples have been gathered, when restricted purely to middle C, or the octave below or above. Other notes are currently excluded to ensure that the samples all contain similar harmonics.

The samples cover many musical instruments, both acoustic, and electronic, including a large number of synthesized sounds.

If needed, the data can be augmented by simply mixing 2 samples, other means may be possible too, but it's important not to distort the spectrum features that we're trying to model.

# Implementation

## Spectograms vs Audio Samples

When working with audio and digital signal processing, an immediate question is whether to work in the time domain with audio samples, or in the frequency domain using the \*Short-Time Fourier Transform\* (STFT) for example.

Algorithms such as the seminal 2016 WaveNet paper by google work in the audio sample space - this has since been significantly improved, and the current state-of-the-art is to work in using audio samples to generate ultra-realistic speech or vocal synthesis.

However, working with samples, at 44.1kHz, requires a huge amount of compute, and it is also more difficult to interpret. The loss function would probably require an STFT to make sense.

I therefore decided for a first foray to work using spectrograms, specifically the STFT.

Other frequency spectrum representations could be considered:

- Mel-Frequency Cepstrum Coefficient (MFCC) could be considered, however these are not suitable for audio reconstruction.

- Constant-Q transforms: these could be useful for analysing music itself where precise note onset and end times are necessary. This is not the scope of the current project.

## Auto-Encoder

An auto-encoder presents itself as a natural choice as we are interested in both sample recreation and new sample generation. The auto-encoder should be able to represent each of the source samples as a unique vector of numbers in a latent space. New samples can then be created in multiple ways: interpolating between samples in that space, or randomly perturbing the vector coordinates of a given sample, or simple generating a random vector.

## Variational Auto-Encoder (VAE)

After some exploration with simple models, I bumped into some well documented problems:

* when interpolating between encoded samples, I would frequently get silence (!!), and occasionally horrible noise.
* the variables in the latent dimensions had no definite range, which also made it difficult to generate new random samples.

This is because the auto-encoded values have no constraints over them, random values might simply point to regions of emptiness which the model has never seen before.

These problems can be resolved using a Variational Auto-Encoder. The encoder effectively produces a distribution of possible encodings, represented as a vector of means and standard deviations - a random sample can be selected from these distributions to represent a given sound. The decoder then learns these distributions and uses this information to recreate the original sample.

Further constraints are imposed on the encoder: the means for each encoded variable should be 0, and the standard deviation 1. This helps ensure that the individual encoded values stay within a reasonable interval, and facilitates interpolation or using the model as as generator.

The mathematics of the VAE are reasonably complex, using Kullback-Leibler divergence (https://en.wikipedia.org/wiki/Kullback‚ÄìLeibler\_divergence) to estimate how far the model's distribution is from an ideal canonical normal distribution with mean 0 and variance 1. In practice the literature provides various tricks to implement this, specifically using the log variance of the distribution. Fortunately, I found many online tutorials and videos explaining the concepts and how to implement a VAE.

## Evolution of the Model

My initial auto-encoder model was simply four fully connected layers, the decoder is implemented as a reverse of the encoder. This allowed me to verify that my entire tech stack was working. Some experimenting revealed that three layers were unnecessary, however four could achieve reasonably good accuracy.

I then started looking at models that could interpolate across the frequency spectrum and across time in a more reasonable manner.

* MLP: at each time-step, an MLP is trained using the previous spectrogram slice (initialised to 0 for the first slice) and the current slice, it outputs a vector of "control" variables. The decoder is a symmetric version of this process, taking in the prevoius generated slice (0s initially) and the control variables, and generates the corresponding spectrogram slice.
* RNN: in a similar way, an RNN is trained at each time step, learning how best to summarise 1 time-step into a set of control variables.
* In practice the RNN is maybe 2 to 3x faster than the MLP and can reach similer accuracy, but with a larger model.
* Variational Auto-Encoder: this is an inner-core that maps from the hidden variables per time-step, to a small latent space. It has not been implemented in such a way that it's reasonably easy to use it inside any other naive Auto-Encoder, for example combining the MLP Auto-Encoder or the RNN Auto-Encoder to create VAE versions of these with much smaller latent spaces.
* GRUs, LSTMs, ...: further models could be explored with better modelling of the evolution of the spectrum over time.

## Complex Numbers...

In a first pass I discarded all phase information, using only the amplitudes. Whilst this reduces the data-size by half, and considerably simplifies the interpretation, it did introduce significant artefacts, even if replaying the original data.

Once I had a working model, I thought I'd convert the inputs to using complex numbers instead. The model was presented with pairs of floats, representing the real and imaginary parts of the numbers. This effectively doubled the data size, but I compensated for that by reducing the frequency range to 11 kHz. Having the phase information does significantly reduce the systemic artefacts dues to zeroing the phases.

However, whilst the base models worked, the more complex variational auto-encoders simply failed to reach a usable accuracy. I spent a long time looking into this, but overall, it is extremely difficult to infer patterns from complex numbers, magnitudes are much more straightforward to deal with. It might be possible in future work to use magnitude and phase instead.

I then discovered the **Griffin-Lim** algorithm which aims to infer plausible phase information purely from the magnitudes. This has greatly reduced the artefacts introduced by discarding phases, so the current model now works using solely phases which has the benefit of being easier to interpret too.

# Training & Hyper-Parameters

Training has proven extremely difficult. As we already know from the practical experience gathered during this course, hyper-parameter tuning is paramount.

To keep things simple, I used skopt's gp\_minimise (https://scikit-optimize.github.io/stable/modules/generated/skopt.gp\_minimize.html). Internally this uses a Matern kernel + Gaussian noise, and provides a configurable search-space for integers, reals and categories, with uniform or log distributions. Generating plots of the hyper-parameter tuning over time has been helpful to prove that it generally works, but I have occasionally found instances where **gp\_minimize gets stuck in sub-optimal local minima**, even when it has already found better points. Using a better optimiser such as TurBO or Optuna would be beneficial.

## Lessons & Observations

* Possibly due to the complexity and size of the training data-set, the hyper-parameter tuning always veers towards the smallest batch sizes allowed. This makes training very slow as we're no longer able to properly leverage the available GPU power (would require multi-threading the GPR).
* It's important to set limits on the model-size as the hyper-parameter optimisation can easily veer to multi-gigabyte models!
* In order to be able to train models with different numbers of layers, I've created a function 'interpolate\_layer\_sizes(start, end, depth, ratio)'. Start and end are usually determined by the problem set, leaving us with 2 hyper-parameters, depth = the number of layers, ratio = the power to use whilst interpolating. This is reasonably flexible and allows me to create models where the layer sizes are large at the beginning, or the end, or flat.
* I don't believe it's possible to reduce the training data-set size when optimising hyper-parameters, because the purpose of this model is to learn as many wave forms as possible. If the training dataset is shrunk, a smaller model may be found by the hyper-parameter tuning that wouldn't then be able to learn the larger data-set.
* I wrote a method to automatically identify the most 'diverse' set of samples from an input vector. This works by finding the sample closest to the average of the training set (so we have the "mid-point" effectively) and then iteratively adding the samples that are furthest away from the average of the subset so far. This has been useful for generating ball-park hyper-parameter estimates but in the final version I had to use the full dataset.
* Stopping conditions can be complex: the Adam optimiser sometimes jumps to a significantly worse loss, then recovers to an overall better loss. I therefore stop if the moving average of the loss is stalled.
* The hyper-parameter optimiser seems to always minimise the weight-decay, this makes sense as the amount of over-fitting is not included in the loss function.
* When stopping the hyper-parametrisation early, it might be worthwhile taking the rate of change of the loss at the point into account, instead of purely the minimal loss achieved: indeed, a model that has stalled will not improve any further, whilst a model that is still improving at 0.5% per epoch may be able to progress much further given more time.

## Over-fitting

Whilst training, an overfitting ratio, defined as test loss / train loss, is computed. For example, if the loss on the training set is half that on the test set, overfit will be equal to two. This gives a good indication of whether the model is going too far and we need to stop training, if overfit > 1.3 for example.

However, for this audio generation problem, it is desirable to reproduce samples as accurately as possible, so overfitting the original training samples may not necessarily be a bad thing.

Over-fitting may also lead to the Auto-Encoder being less able to generalise to samples outside the original training dataset.

## Incremental Training

The models I'm using are a combination of 2 models: a naive auto-encoder on the outer layer, followed by a VAE. The outer layer typically compresses the data by a factor of 6. The inner VAE layer compresses the layer by a factor of 3000 to 4000, down to a small number of latent variables.

In practice, training the two models simultaneously proves intractable:

Naive-Encoder -> VAE-Encoder -> Latent Space -> VAE-Decoder -> Naive-Decoder

The model is simply too deep with possibly over 15 layers. The hyper-parameter tuning space is also the product of the spaces for each individual model.

As the performance of the end-to-end VAE will be gated by the performance of the outer naive encoder, I decided to split the training as follows:

1. Find the best hyper-parameters for the outer "naive" auto-encoder.
2. Train the "naive" outer auto-encoder with longer time scales using the best hyper-parameters
3. Find the best hyper-parameters for the inner VAE using a frozen set of optimal parameters for the outer auto-encoder.
4. Train the inner VAE with longer time scales using the best hyper-parameters

This has several benefits:

* We can optimally tune the outer auto-encoder, with the RNNs, MLPs, LSTMs etc.
* We can then train the VAE independently.
* A huge performance speed-up (ie: over 100x): we can use encode versions of the train & test datasets. Instead of simply freezing the layers we don’t want to change on the entire model, we encode once all of the sample STFTs and then train the VAE against these encodings.
* Importantly: the end-to-end model training would fail because it was too complicated (disappearing gradients etc.)
* The search space for the hyper parameter tuning is significantly reduced. If the outer model had a search-space of N parameters, and the inner VAE had a search space of M, the original search-space would be N \* M, whilst it is now N + M. (i.e.: 9 vs 6 in my use-case, this makes a huge difference!)
* We can use different hyper-parameters for the optimiser (Adam: learning rate, batch size, weight decay) for the VAE than for the outer auto-encoder.

# Tests

## Sound Quality

Identifying a suitable loss metric for audio is complex. Converting to a log scale such as decibels would be a reasonable first pass, but that would further complicate the gradients in the complex STFT case.

Ultimately, I'm simply using the MSE Loss between the original and regenerated spectrograms, and whilst this has no human perceptual meaning, it does allow the models to converge reasonably.

In addition, I have implemented mu-law encoding to emphasise the louder parts of the spectrum. However, it is also very important to capture the "decay" or "tail-off" portions of sounds: scaling a sound by a factor of 100 is perceived as the same sound, just not as loud. The human ear accommodates for a factor over 100,000 in amplitudes! (The internal ear mechanisms include a variable gain adjustment, which is why sudden loud sound will be painful if unexpected).

Ultimately the best test is simply human perception, and at present the models don't sound that great anyway, so this isn't really a tough issue.

## VAE Distribution

I've created plots of all the variables in the latent space: individually, in pairs, and for selected sub-sets of sample types, for example pianos vs strings.

This has been helpful to highlight the size of the latent space required (sometimes a variable will be barely used), and that generally we are achieving our ideal mu=0, std=1 distribution for each variable.

Surprisingly, when hyper-optimising, there was little difference in accuracy when the latent space was 5 or 11 variables. However, on examining the plots of the encodings, it became apparent that some of the variables were not actually be used much and had a tiny stdev. So it would seem that a 5 (or possibly 6) dimensional latent space is sufficient!

# Key Lessons Learnt

The key points here are very similar to my work on the CapStone hyper-parameter optimisation problem:

- You need to know in detail what's the code is doing. I have stumbled on so many gotchas which can be sometimes masked by the capabilities of the deep neural networks used. For example, it took me a long time to resolve an artefact that was introduced by the layout of the real & imaginary numbers in memory.

- ChatGPT-4 is an invaluable help as a way of prototyping code or getting advice on how to approach problems.

- Data Visualisation is really important: this helps spot so many problems or prove that things are working as you'd expect them to.

- Variational Auto-Encoders are a necessity: I quickly (re-)discovered why naive auto-encoders were not sufficient.

- It's unnecessary to add a weight to the KL-Divergence term in the VAE loss function: it seems the optimiser is always able to force the distributions into mu=0, stdev=1.

- YouTube tutorials can also be very helpful. There are long series (which I've partially watched) on audio synthesis in particular, music generation etc.

- Working with Audio is significantly different from working with Images. CNNs do not work well with Audio, nor does max-pooling apply well. An interesting article on Medium explained that audio is "transparent", unlike objects in an image which occlude each other. If you play two sounds together, you hear two sounds (in most cases), one sound does not hide the other, whilst in images each pixel can generally be attributed to a single object. There are also complex scaling issues: a sound at a 100x smaller magnitude is still perceived as the original sound. The human ear can accurately hear sounds spanning over 100 dB in range, ie: 5 orders of magnitude!

## Future work

The key item to work on is better modelling in the time-domain. I don't know whether LSTMs or some other transformer model will crack this, but this is the current gating factor.

A related issue: it would be brilliant to have a model that was independent of the sample length, using some sort of time dilation.

Increasing the sample dataset would be beneficial, particularly with more synth, vocal and rhythmic sounds. Generating basic synth sounds could be implemented algorithmically.

The model has many possible uses:

* morphing between samples with randomisation to generate new sounds,
* as a back end for a "text to audio" generator,
* sample classification using the encoder,

There's also way more to model:

* different pitches: including how the timbre changes as you move up and down the note range.
* different note velocities: modelling how the timbre changes on acoustic instruments played pianissimo to fortissimo.
* modulations: vibrato on violin and voice for example.

Ideally, I would like to shift to working using audio samples rather than spectrograms, that appears to be what is used in all the state-of-the-art models and publications.

# Conclusion

This has been a huge and time-consuming project. It wouldn’t have been possible without having my own GPU-accelerated hardware. The overall results are sadly not as good as I would have liked and will require more work. Overall, whilst I've learnt a lot implementing all of this myself, I've also got a much better appreciation of how much more I need to learn!