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.

SAMPLE 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 many 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.

The sample names have been used to automatically assign an approximate category to each sample, yielding the following approximate distribution:

```
Loaded 1017 STFTs from STFT 44100 Hz, size=1024, hop=1024.pkl
Samples by Category:
 231 =  22.7% : Vocal
 176 =  17.3% : Synth
 127 =  12.5% : Guitar
 119 =  11.7% : No Category
  62 =   6.1% : Bass
  62 =   6.1% : Synth Makes
  49 =   4.8% : Plucked
  31 =   3.0% : Bell
  30 =   2.9% : Strings
  29 =   2.9% : Pad
  23 =   2.3% : Piano
  20 =   2.0% : Brass
  19 =   1.9% : E-Piano
  17 =   1.7% : Wind
 8 =   0.8% : Organ
 8 =   0.8% : Percussion
 6 =   0.6% : Pulsing
```

IMPLEMENTATION

SPECTROGRAMS 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.

DATA PRE-PROCESSING

After several rounds of experimentation, the following pre-processing is used:

- Trim or pad the input sample to the desired duration (currently 2 seconds)
- Normalise the sample to [-1, 1]
- Convert to magnitudes, discarding the phase information (this is a big compromise).
- Remove magnitudes less than -70 dB
- Use mu-law encoding with factor = 8. This prioritises low amplitude sounds whilst losing compressing the louder parts of the signal.

This transformed data is fed to the model.

When converting back to audio, the same steps are used in reverse, but in addition, the **Griffin-Lim** algorithm is used to recover plausible values for the phases. Without this correction, the reconstituted sound has severe artefacts.

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 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.

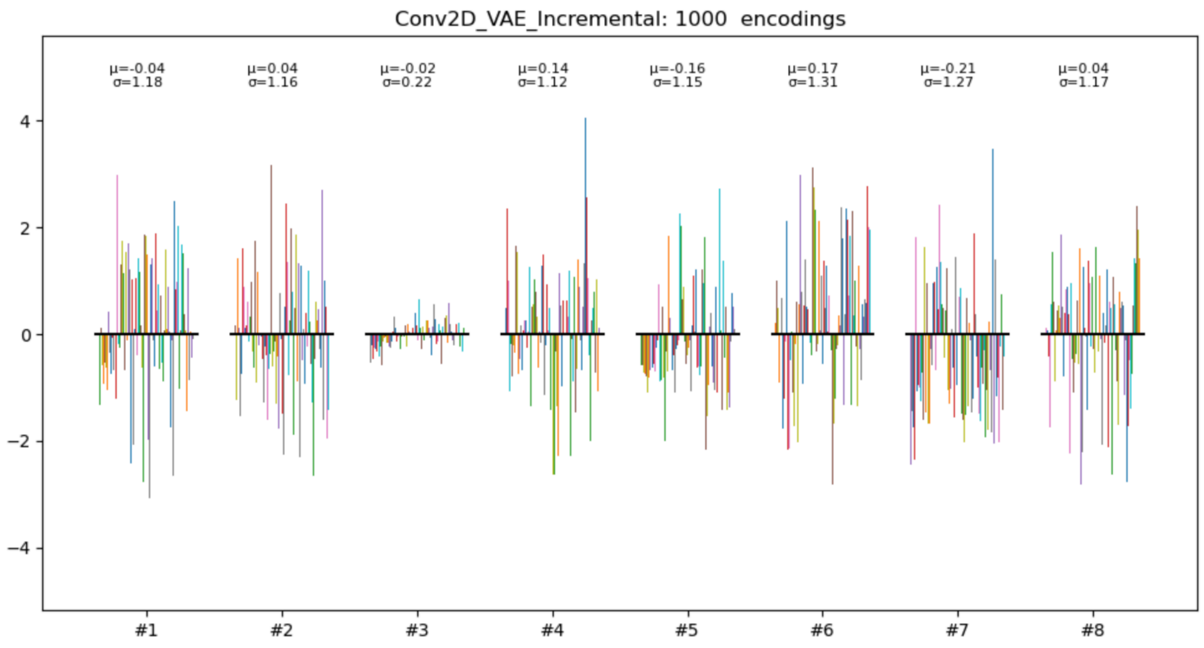
DIMENSION OF THE LATENT SPACE

Although I have use various hyper-parameter optimisation methods to establish a reasonable value for the size of the latent space, I found that the model accuracy is relatively unaffected by this number, if it is large “enough”.

If we assume that each dimension encode N levels, and we have S samples to encode, then S^(1/N) is roughly the number of variables one would expect.

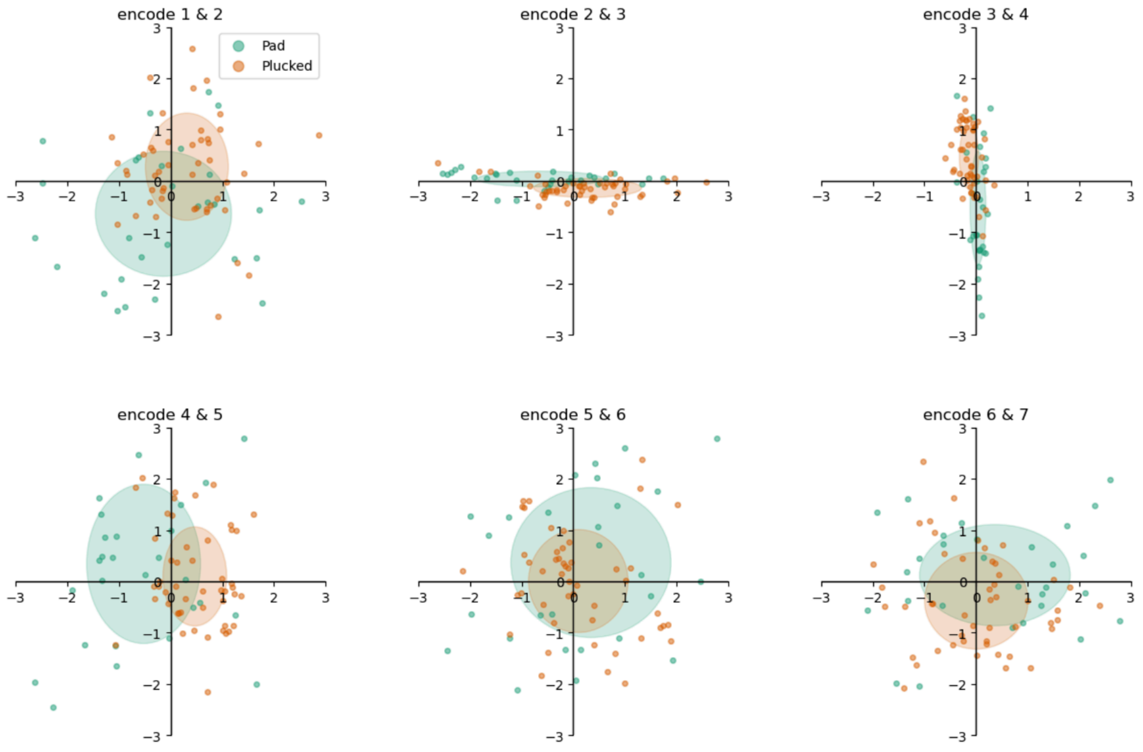
In our case, using S=1000 samples, and N=4, we obtain latent = 6, which is a surprisingly small number.

Hyper-parameter tuning confirms that this value works, 5 tends to be a little too small, whilst larger numbers such as 8 lead to under-utilised variables, as illustrated below:



XY plots of pairs of variables reveal how variable 3 is not terribly useful in this model:

```
Encoded samples:
  29 x Pad
  49 x Plucked
```



In this figure we can see the encodings of 1000 samples, the optimiser has done a reasonable job of attaining mean=0 and stdev=1 for each variable, except variable 3. In practice reducing the latent space from 8 to 6 works equally well.

MODEL SIZE

It is important that our model should be able to generalise to new samples that it hasn't seen before. I'm particularly wary of overfitting the model.

We know that our input data-set is 1,017 samples x 1,025 frequency magnitudes x 83 time-steps.

Each sample is represented as 85K values, our entire data set is 86M values. I have therefore limited the model size to 20M values in total, ie: 10M each for the encoder and the decoder. This helps ensure that the model doesn't purely learn the training data-set as it is significantly constrained.

In practice the final model size is approximately 8.6M parameters in total for the combined encoder + decoder.

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 similar accuracy, but with a larger model.

- Sequence of **2D convolutions**: as the spectrogram is a (black & white) 2D image it makes sense to consider 2D convolutions to extract key features from the training data set. However some care must be taken as blindly using 2D convolutions would effectively blur the signal both in time and space. The first layer of the encoder, and last layer of (symmetrical) decoder use a stride of 1 to help ensure we don't have a 'stuttering' effect when resynthesizing the sound. In addition I deliberately limited the maximum compression of the 2D convolutions. The space compression comes from the stride, ie: stride=(2, 2) will reduce the size of the input by a factor of 4.

- 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.

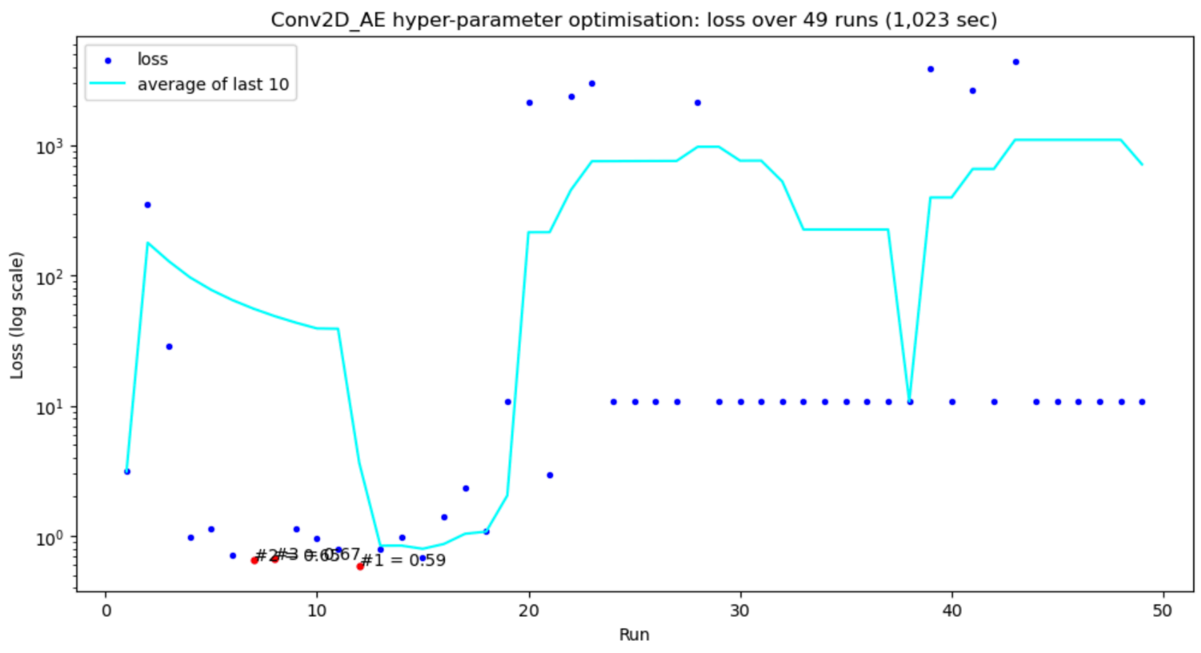
TRAINING & HYPER-PARAMETERS

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

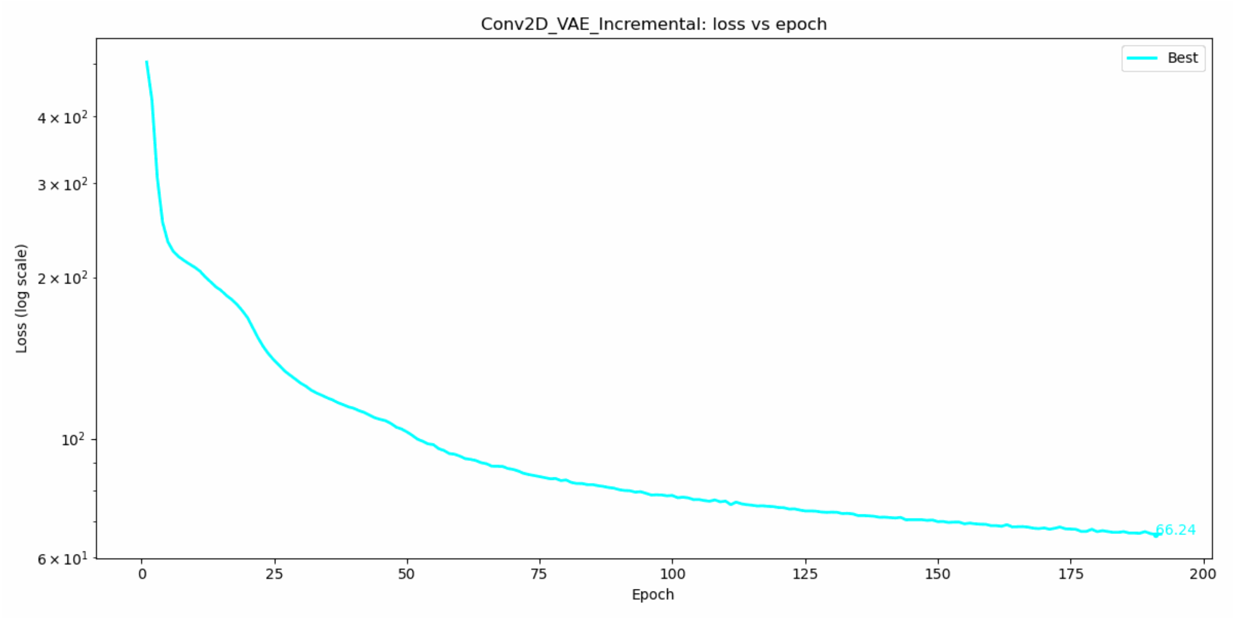
GAUSSIAN PROCESS REGRESSION

To keep things simple, I used SciKitOpt's gp_minimise. 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.

Here's an example of gp_minimize failing to explore around the original low points it found early on, and getting stuck retrying some local minimum many times.



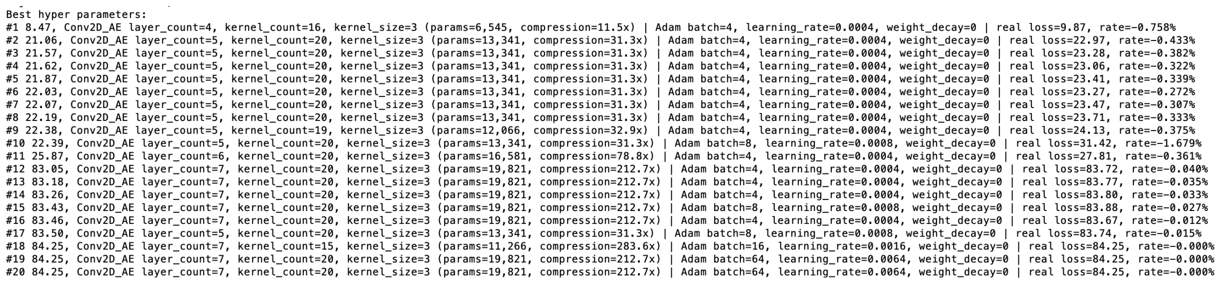
The following animation shows the progress of the hyper-training over time. Note that it uses an early cut-off, if at a given epoch a model is 3 times worse than best known model at that epoch, we stop the training.



GRID-SEARCH

As the models used here have a small number of hyper-parameters (typically 3) I found **grid-search** to be a practical alternative, yielding better results whilst ensuring all bases are covered. Ultimately this is what I have used to hyper-tune the auto-encoder and the variational auto-encoder.

Example grid-search output:


```
Best hyper parameters:
#1 8.47, Conv2D_AE layer_count=4, kernel_count=16, kernel_size=3 (params=6,545, compression=11.5x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=9.87, rate=-0.758%
#2 21.06, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=22.97, rate=-0.433%
#3 21.57, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=23.28, rate=-0.382%
#4 21.62, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=23.06, rate=-0.322%
#5 21.87, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=23.41, rate=-0.339%
#6 22.03, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=23.27, rate=-0.272%
#7 22.07, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=23.47, rate=-0.307%
#8 22.19, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=23.71, rate=-0.333%
#9 22.38, Conv2D_AE layer_count=5, kernel_count=19, kernel_size=3 (params=12,066, compression=32.9x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=24.13, rate=-0.375%
#10 22.39, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=8, learning_rate=0.0008, weight_decay=0 | real loss=31.42, rate=-1.679%
#11 25.87, Conv2D_AE layer_count=6, kernel_count=20, kernel_size=3 (params=16,581, compression=78.8x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=27.81, rate=-0.361%
#12 83.05, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=83.72, rate=-0.040%
#13 83.18, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=83.77, rate=-0.035%
#14 83.26, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=83.80, rate=-0.033%
#15 83.43, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=8, learning_rate=0.0008, weight_decay=0 | real loss=83.88, rate=-0.027%
#16 83.46, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=4, learning_rate=0.0004, weight_decay=0 | real loss=83.67, rate=-0.012%
#17 83.50, Conv2D_AE layer_count=5, kernel_count=20, kernel_size=3 (params=13,341, compression=31.3x) | Adam batch=8, learning_rate=0.0008, weight_decay=0 | real loss=83.74, rate=-0.015%
#18 84.25, Conv2D_AE layer_count=7, kernel_count=15, kernel_size=3 (params=11,266, compression=283.6x) | Adam batch=16, learning_rate=0.0016, weight_decay=0 | real loss=84.25, rate=-0.000%
#19 84.25, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=64, learning_rate=0.0064, weight_decay=0 | real loss=84.25, rate=-0.000%
#20 84.25, Conv2D_AE layer_count=7, kernel_count=20, kernel_size=3 (params=19,821, compression=212.7x) | Adam batch=64, learning_rate=0.0064, weight_decay=0 | real loss=84.25, rate=-0.000%
```

This makes is easier to see how the number of layers increases the accuracy, whereas the latent size does not matter much.

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 encoded versions of the train & test datasets. Instead of simply freezing the layers we don't want to change on the entire model, we encode all of the sample STFTs once and then train the VAE against these encodings.
- Importantly: the end-to-end model training would fail because the network is too complicated and deep, leading to 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.

CURRENT "BEST" MODEL

Following many weeks of hyper-parameter training and model training, the following model is the best found so far.

Encoder: input STFT → [5 x Conv2D layers] → [4 Fully-Connected + ReLU layers] → 7 Latent variables.
Decoder: 7 latent variables → [4 Fully-Connected + ReLU layers] → [5 Conv2DTranspose layers] → output STFT

```
model=CombinedVAE(
  (auto_encoder): Conv2DAutoEncoder(
    (encoder): Sequential(
      (0): Conv2d(1, 20, kernel_size=(4, 4), stride=(1, 1))
      (1): Conv2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (2): Conv2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (3): Conv2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (4): Conv2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
    )
    (decoder): Sequential(
      (0): ConvTranspose2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (1): ConvTranspose2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (2): ConvTranspose2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (3): ConvTranspose2d(20, 20, kernel_size=(2, 2), stride=(2, 2))
      (4): ConvTranspose2d(20, 1, kernel_size=(4, 4), stride=(1, 1))
    )
  )
  (vae): VariationalAutoEncoder(
    (encoder_layers): Sequential(
      (0): Linear(in_features=6300, out_features=1850, bias=True)
      (1): ReLU()
      (2): Linear(in_features=1850, out_features=1008, bias=True)
      (3): ReLU()
      (4): Linear(in_features=1008, out_features=443, bias=True)
      (5): ReLU()
    )
    (fc_mu): Linear(in_features=443, out_features=7, bias=True)
    (fc_logvar): Linear(in_features=443, out_features=7, bias=True)
    (decoder_layers): Sequential(
      (0): Linear(in_features=7, out_features=443, bias=True)
      (1): ReLU()
      (2): Linear(in_features=443, out_features=1008, bias=True)
      (3): ReLU()
      (4): Linear(in_features=1008, out_features=1850, bias=True)
      (5): ReLU()
      (6): Linear(in_features=1850, out_features=6300, bias=True)
    )
  )
)
```

CODE STRUCTURE

The code has been split into multiple modules as follows:

Utilities:

- AudioUtils.py: utility functions for computing STFTs, Mu-Law, Decibels etc.
- Augment.py: generates new training samples by randomly mixing two samples from the train dataset.
- MakeSTFTs.py: converts samples into STFTs and pre-processes them for us in the model, and post-processes them back to audio.
- Debug.py: displays variables
- Device.py: run the model to CPU, GPU or MPS (Metal Performance Shader on Mac).
- SampleCategory.py: infers a sample's category based on its file name, approximate.
- Graph.py: various plots, histograms, bar-charts, spectrograms etc.

Models:

- ModelUtils.py: utilities for constructing models, or calculating the number of parameters in a given model.
- MakeModels.py: instantiates models given a name and a set of hyper-parameters.
- Conv2D_AE.py: an auto-encoder using 2D convolutions.
- MLP_AE: an auto-encoder using a time-based multi-layer perceptron.
- VariationalAutoEncoder.py: implements a simple VAE using fully-connected layers. Also provides a class CombinedVAE so which can wrap any standard auto-encoder into a VAE. Both can be trained separately with huge performance benefits.

Training

- HyperParameterTuning.py: this is the top-level entry point for training models. It includes both GPR and grid-search algorithms to optimise the model's parameters.
- Train.py: the core model training code, keeps track of test & train losses, and can terminate early if the model is performing poorly compared to the best known so far.
- Generate.py: utilities to create new sounds combining existing samples and the model.

Many of the files can be run stand-alone and will execute various tests, or train models.

The code can also be run in a Jupyter notebook which is helpful for auditioning sounds and viewing graphs.

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.

ACCURACY

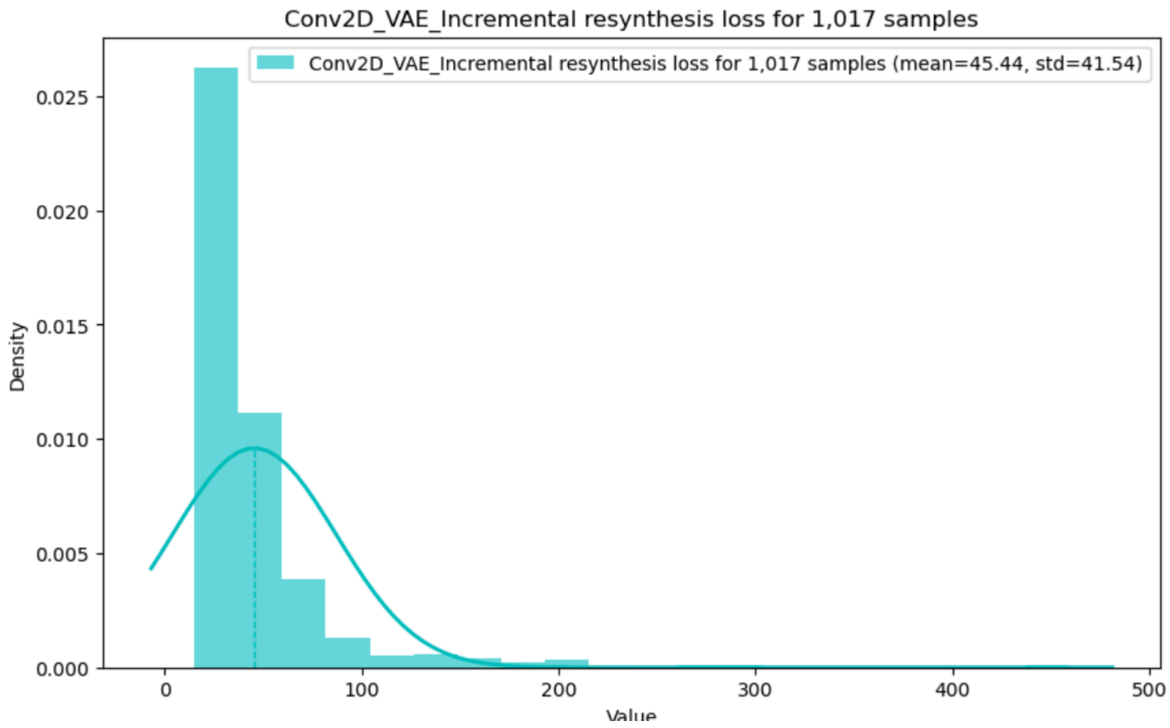
Loss is measured as the Mean Square Error between the input and output spectrograms when running a sample end-to-end through the model.

The loss L is not normalised by the size of the input, 85K samples in this case, we can express the standard deviation as a % using the following formula:

stdev% = 100 x sqrt(loss/85,000).

OVERALL END-TO-END

The chart below shows the loss across all the input samples, there are some nasty outliers, and the mean loss is 45.4, which equates to approx. 2.2%. Losses of < 1% start sounding reasonable, ie: a 4 times improvement with a reported loss < 10.



CONV2D AUTO-ENCODER

The Conv2D auto-encoder achieves a much better loss of **16.4** with no over-fitting (test/train = 1.02).

This indicates that Variational Auto-Encoder is having trouble fully reconstituting the features output by the Conv2D encoder.

COMPRESSION RATIOS

2D Convolution: the current best model, compresses the input STFT by a factor of **13.5**:

```
Conv2D_AE layer_count=5, kernel_count=20, kernel_size=4 (params=13,621, compression=13.5x)
```

The input STFT has size 1025x83 = 85,000 and is internally compressed down to 20 x 63 x 5 = 6300 achieving a compression ratio of **13.5** (the input spectrogram of 1024x83 is shrunk to 20 features of 63x5 each).

The VAE then compresses its input from 6300 down to 7, achieving a compression ratio of **900**.

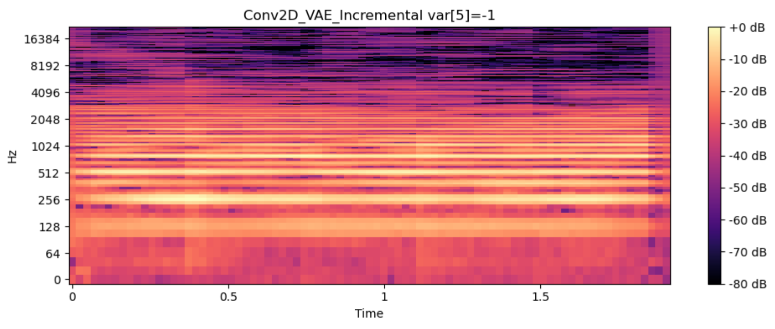
```
VariationalAutoEncoder: layers=[6300, 1850, 1008, 443, 7], parameters=27,954,907, compression=90
0
```

PERCEPTUAL TESTS

Whilst the sounds coming out of the model are somewhat recognisable, there a number of draw-backs:

- Too little differentiation: the model tends to generate 'average' sounds that are insufficiently differentiated.
- Noisy tails: the regenerate spectra often contain high frequency noise that isn't present in the original input.
- Stuttering: the output appears blocky in the time dimension due to the expansion of the convolutions. This is visible in the spectrograms.
- High frequency phasing noise: this is probably an artefact discarding the phase information and trying to infer it using the Griffin-Lim algorithm.
- Imprecise harmonics: whilst the lower harmonics are well reproduced, the upper harmonics seem to be become mushy. This only matters if aiming for high quality audio. Well-known spectral synthesis methods would typically replace the upper harmonics with the appropriate amount of white noise!

Clear example of blocks in the time dimension: these are audible as a stuttering effect.



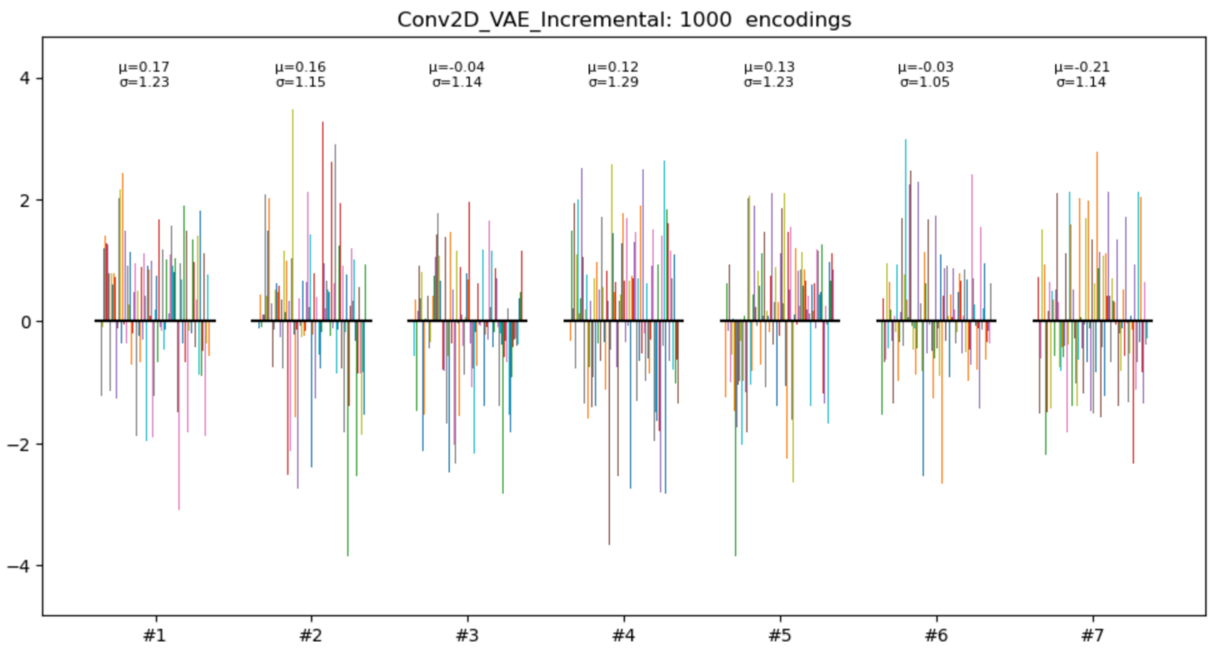
Smoothing in the time dimension could reduce these artefacts but would further blur the sound.

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.

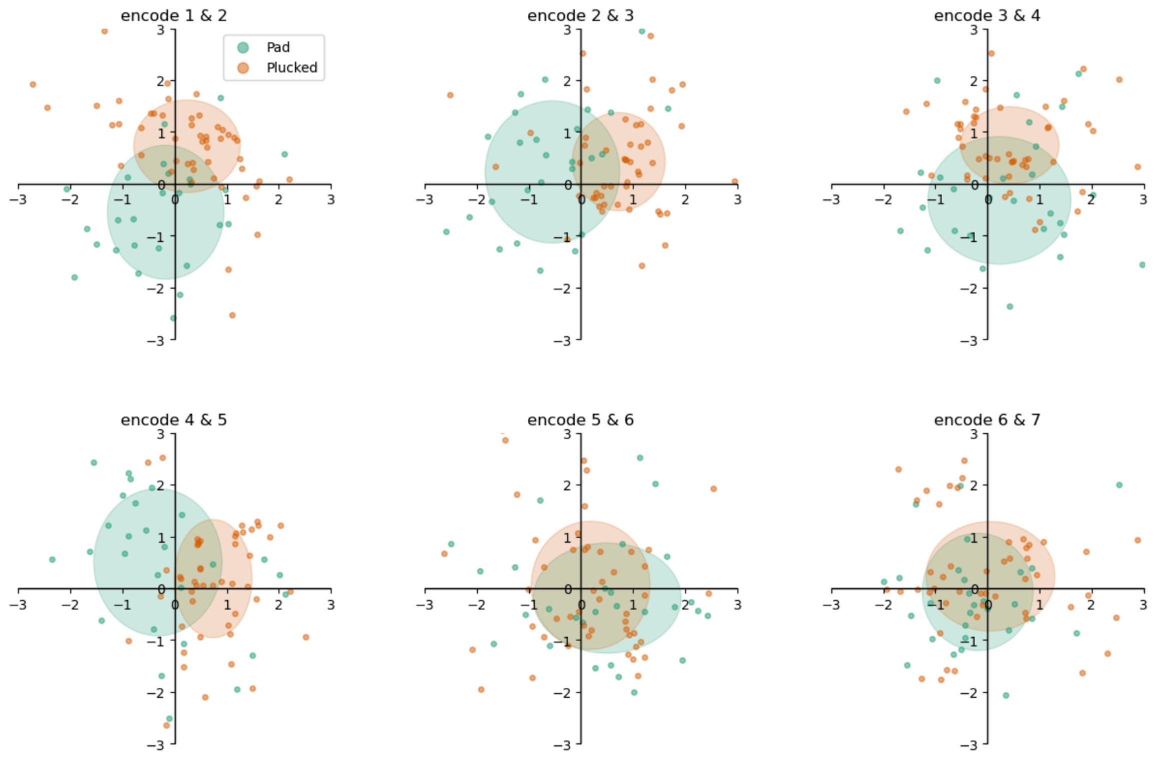
This has been helpful to highlight the size of the latent space required (sometimes a variable will be barely used), and that generally we are achieving our ideal mu=0, std=1 distribution for each variable.

The following chart displays a 1000 encodings for a latent space of 7 variables. It also computes the mean & stdev for each variable, showing how the model has approximately converged to the normal Gaussian distribution.



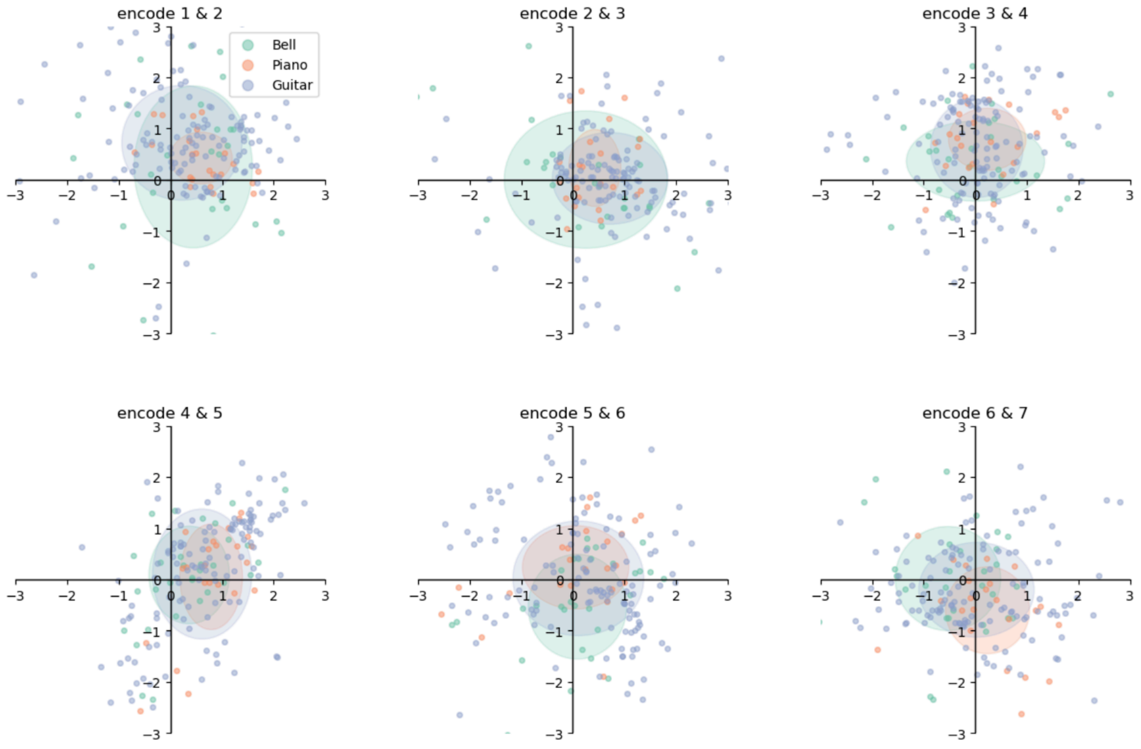
The following is a scattergram for “pads” (typically long sustained sounds) vs “plucked” sounds (short). Although we don’t know what each of the dimensions is encoding, we can see a clear differentiation on some of the variables.

```
Encoded samples:
  29 x Pad
  49 x Plucked
```



This is the same model for “Guitar”, “Piano” and “Bell” sounds, the plots overlap each other much more:

```
Encoded samples:
  31 x Bell
  23 x Piano
 127 x Guitar
```



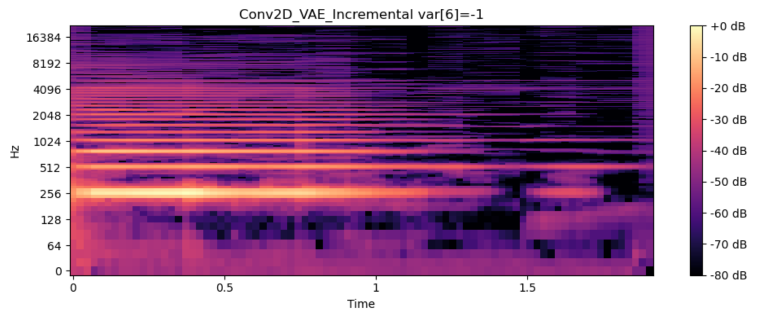
This may make sense in that they are all sounds that start with lots of high frequencies and then decay over time.

Note: the sample categories are approximate, I assigned them by processing the sample names. It could be interesting to use the model as a sample classifier, or to perform some clustering based on the encoded values in the latent space. VAE’s can also be used for outlier detection.

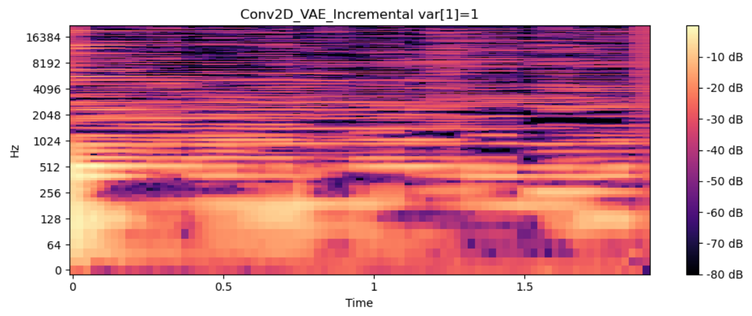
GENERATING SOUNDS

This is the fun part of the project. I’ve implemented the following methods to generate sounds:

- Try to understand the encodings: set all variables but one to zero and generate a sample. This gives us some idea of what features the VAE has encoded internally. These sounds can be quite interesting, and reminiscent of some of the samples in the training set.



In this example we have a recognisable “guitar pluck” sort of sound, the latent space input is [0, 0, 0, 0, 0, -1, 0].



This example is just plain ‘odd’ – a lot of frequencies represented with odd pockets of blanks.

- Run a sample that is not in the data set through the auto-encoder. As the VAE is non-deterministic (random numbers are drawn during the internal reparameterisation) we can run this multiple times and obtain slight variations. In practice this produces a type of **style transfer**: if I input my voice singing “Baaah” the output is reminiscent of a guitar sample with vocal overtones.

- Generate samples around a given input sample from the training set, with various amounts of noise added to the latent space. As the noise level increases, we deviate more and more into other strange sounds.

- Interpolate linearly between two samples in the latent space. The start and end should be identical to the input samples (roughly) but in-between we should transition smoothly from one to the other.

KEY LESSONS LEARNT

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

- You need to know and fully understand in great detail what 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 audio artefact that was introduced by the memory layout of real & imaginary numbers. I also discovered that simply adding “nn.Sigmoid()” in the decoder (because the results should always be in the interval [0, 1]) can completely kill a model’s ability to converge.

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

- ChatGPT-4 is an invaluable help as a way of prototyping code, or getting advice on how to approach problems, or even simply best practices for Python.

- 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, just not so loud. 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 a clear 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

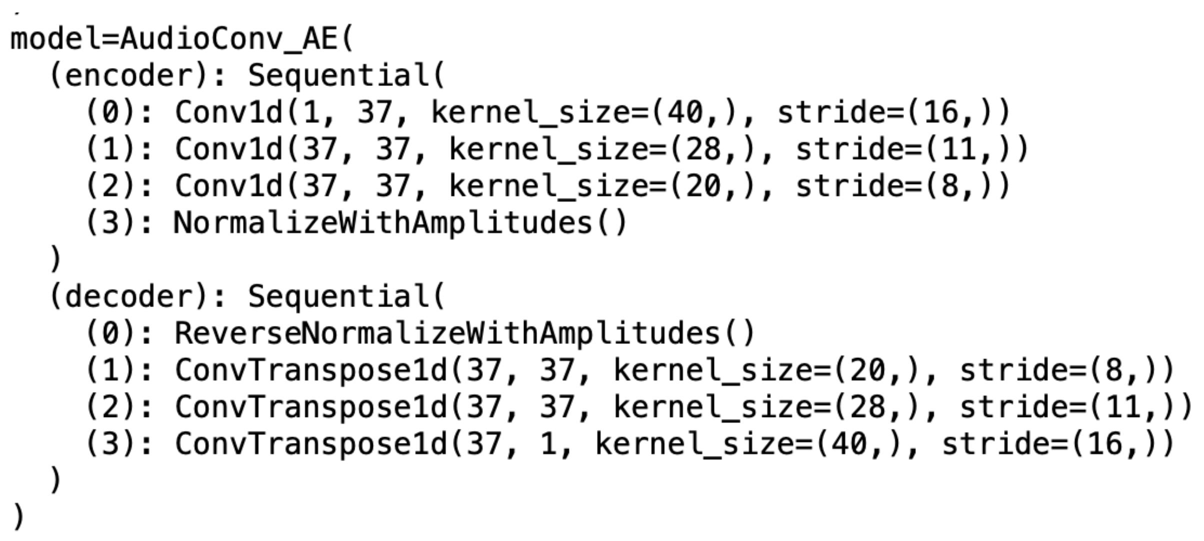
This has been a huge and time-consuming project. It wouldn't have been possible without having my own GPU-accelerated hardware. The overall results are sadly not as good as I would have liked and will require more work. Overall, whilst I've learnt a lot implementing all of this myself, I've also got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

ADDENDUM – JAN 2024

Having continued working on this project since the original submission on 23rd Dec 2023, I've made great process in working with raw samples rather than using STFTs. This has a major benefit that there are no major audible artefacts due to the audio encoder itself. That said (I haven't measured this yet), I think the high frequencies may be trimmed off, ie: similar to applying a low-pass filter.

The code changes for this can be found in the branch named **Amplitude_AE**.

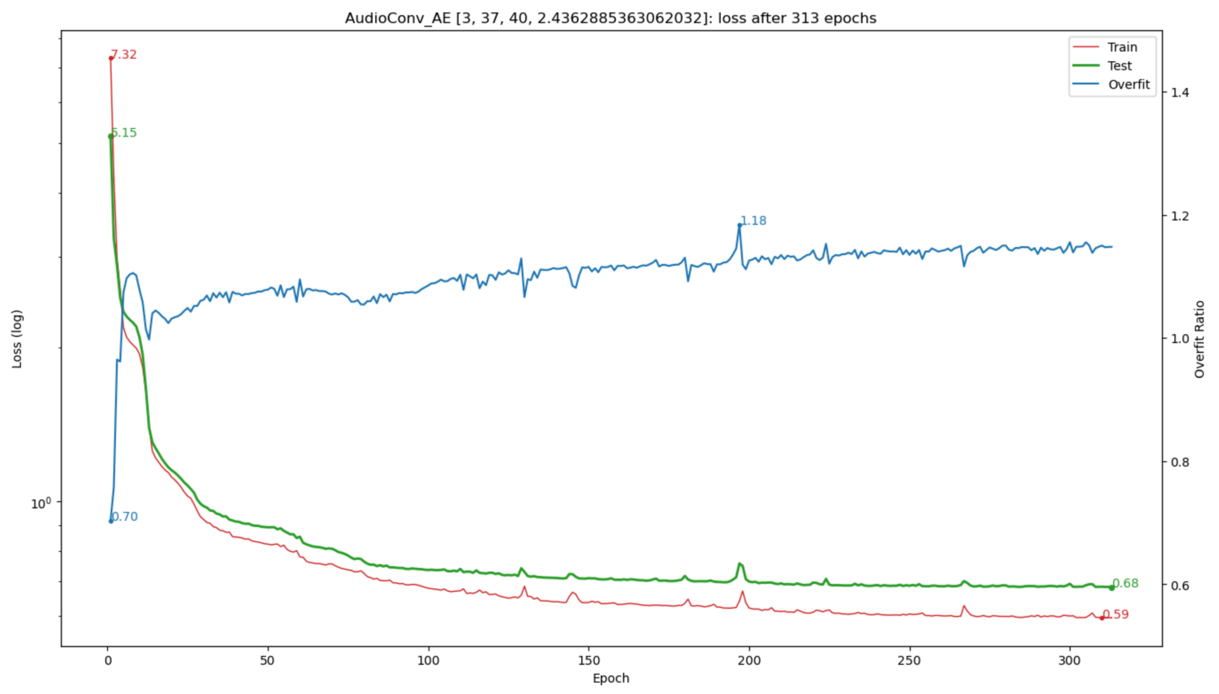
The audio auto-encoder consists of three 1D convolution networks, followed by a last step that normalises the resulting kernels, and stores the necessary amplitude for each. This makes it substantially easier for the VAE to then identify patterns in the data.



```
model=AudioConv_AE(
  (encoder): Sequential(
    (0): Conv1d(1, 37, kernel_size=(40,), stride=(16,))
    (1): Conv1d(37, 37, kernel_size=(28,), stride=(11,))
    (2): Conv1d(37, 37, kernel_size=(20,), stride=(8,))
    (3): NormalizeWithAmplitudes()
  )
  (decoder): Sequential(
    (0): ReverseNormalizeWithAmplitudes()
    (1): ConvTranspose1d(37, 37, kernel_size=(20,), stride=(8,))
    (2): ConvTranspose1d(37, 37, kernel_size=(28,), stride=(11,))
    (3): ConvTranspose1d(37, 1, kernel_size=(40,), stride=(16,))
  )
)
```

The hyper-parameters for this audio compressor were optimised using GPR. The audio encoder has 134,570 parameters and compresses the sound by a factor 39.1.

The audio-encoder achieves less than 0.6% error on the audio samples, with less than 15% overfit.



VAE

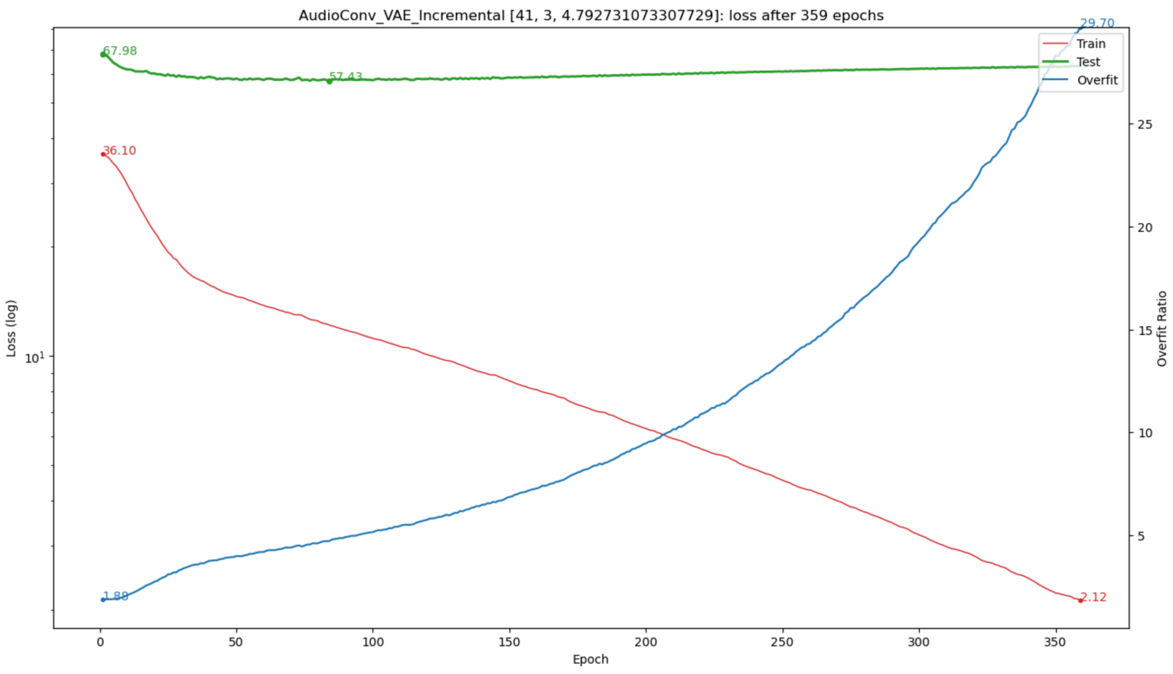
Unfortunately, the variational auto-encoder is unable to converge accurately on this data, it suffers from "posterior collapse" where the decoder ignores the latent variables and produces a single "average tone", whilst all the latent variables converge to 0, with stdev=0. This is a well-documented problem, see https://openreview.net/pdf?id=r1xaVLUYuE for example.

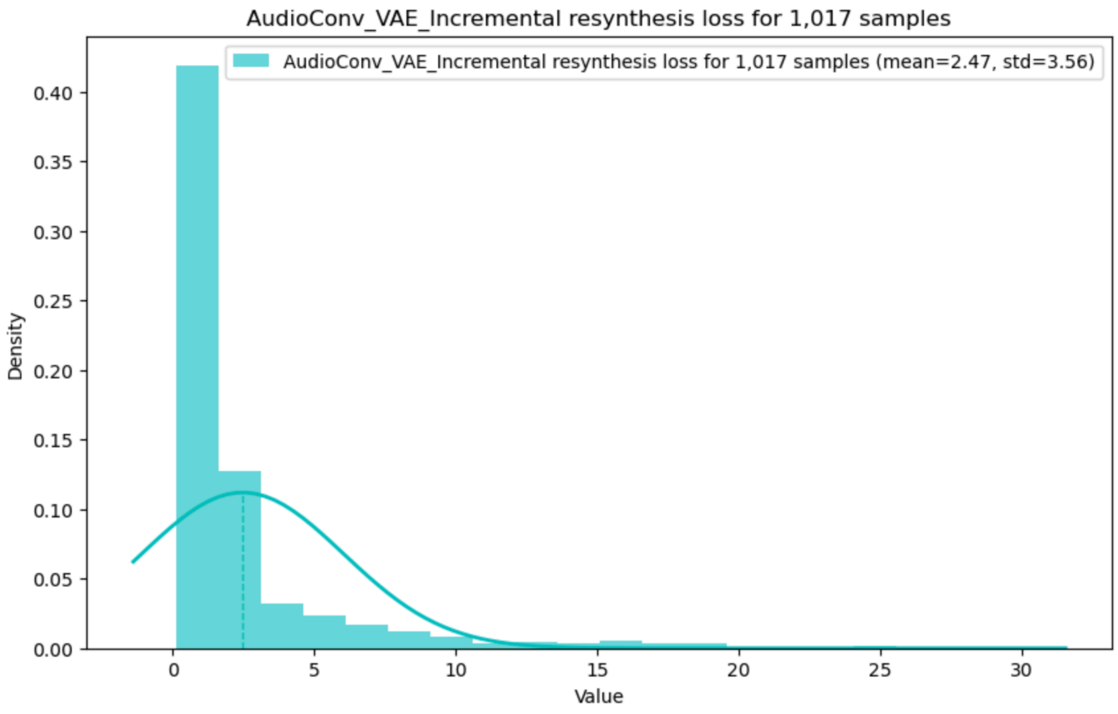
I have tried using annealing to help the VAE converge, but this hasn't yielded satisfactory results. There is a clear trade-off between the VAE and optimal reconstruction of the audio signal. The VAE prevents over-fitting, but is unable to accurately reproduce any single sample, producing a sort of average "orchestral" sound instead (as if all the instruments were playing simultaneously). **The MSE stabilises at 12.5% at best which is totally unusable.**

NAÏVE AUTO-ENCODER

Having established that the VAE was not going to work, I sought a quick alternative: simply using a deep auto-encoder with a latent space normalised to [-1, 1]. It is expected that this solution will lead to "dead zones" in the interpolation between latent vectors.

The naïve auto-encoder is able to accurately reproduce the samples, withing 2.5% MSE, but with **considerable over-fitting (by a factor of 30!)**. A further hyper-parameter search will be required in order to see whether this can be minimised whilst still maintaining good accuracy. Overfitting may be acceptable if the goal is to reproduce the original samples as accurately as possible, however it is not desirable if the goal includes being able to encode new samples.





QUANTISED VARIATIONAL AUTO-ENCODER

The next option to try here is a Quantised Variational Auto-Encoder, which would record a number of latent space vectors that can then be used to encode and decode the signal. I have not implemented this yet.

Work continues apace!