# Clustering Elephant Rumbles

Ben Ellis

October 4, 2021

## 1 Introduction

Computational analysis of human language has progressed quickly recently, with tools such as smart speakers, digital assistants and translation tools becoming ubiquitous. However, these advances have not brought about similar progress in the field of understanding animal communication, where learning must be done end-to-end from recordings rather than text, and any hypothesis must be confirmed by experiment. Although historically data collection in this field was very difficult, the recent development of cheap and reliable sensors has made instead the processing of data a barrier to progress, with many techniques requiring extensive manual labelling.

In this work we focus on the communication of African Elephants (*Loxodonta africana*). Elephants use many different vocalisations to communicate with one another, such as trumpets and infrasonic rumbles[7]. It is these infrasonic rumbles that are our focus in this work. The rumbles have both an acoustic and a seismic component. It is not known whether these components are redundant or contain separate information, and what advantage there is to having these two channels of communication[6]. There is evidence, however, that there is some meaning derived from just the seismic component from playback experiments[5]. In this work we attempt to gain some insight into the content of these infrasonic rumbles by performing unsupervised learning on the rumbles. We use data collected from a 3-week deployment of 22 seismometers as well as a number of microphones and camera traps. This generated a total of 135 GB of seismic data for analysis. It is important to note that three weeks is not a long time in zoological terms – it does not capture any seasonal variation, nor any lifetime effects such as aging. Similarly, 135 GB is not *that* much data. The audio and camera trap data collected for this deployment consume much more space. However, as we shall see, even this relatively short deployment presents significant engineering challenges in the processing of data.

We present a pipeline for the analysis of elephant rumbles with minimal supervision. This pipeline has three important components:

1. The data loader
2. The detector

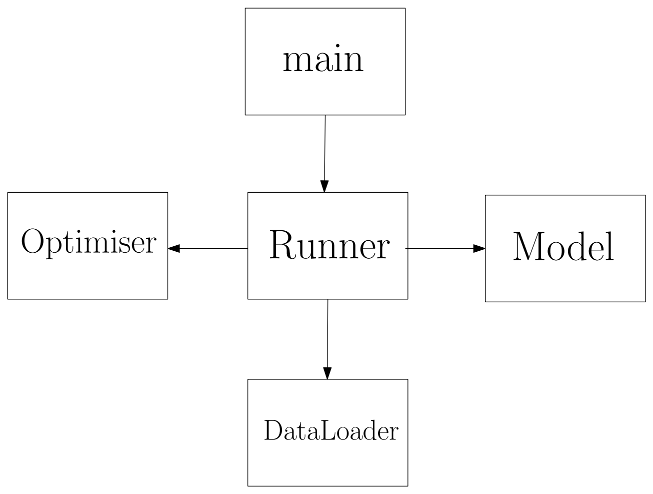

Figure 1: The architecture of a main component of the rumble processing tool.

3. The clusterer

The data loader has the seemingly simple job of loading data from disk and serving it to whatever component performing downstream analysis requires it. The detector's goal is to identify rumbles in the seismic data which can then be fed to more complex analysis. Training the detector requires some manual analysis of the data to provide labels for the supervised training of the detector, however a relatively small selection of data can be used. The third component, the clusterer, takes as input identified rumbles, and groups these together in a meaningful way. In the rest of this report, each component of this pipeline is discussed in turn. However, before that I outline some important aspects of the software itself.

## 2 System Architecture

The guiding principle of the design of the code of this pipeline was that it should be *modular*. This was important because it was likely that different parts would be found to be inadequate and therefore have to be replaced or greatly rewritten. Without effective modularisation this could prove difficult. Therefore the detector and the clusterer are implemented as separate versions of the architecture shown in Figure 1. The major components of this architecture have the following functions:

1. **main**. This is the main loop of the system, responsible for injecting items into the config object, creating the important components and looping through epochs.

2. **Runner**. This class is responsible for the training and testing loops of an epoch. This gets batches from the dataloader, feeds them into the model, evaluates the loss, and calls the optimiser to update the weights.

3. **DataLoader**. This is an implementation of PyTorch's `DataLoader` class. This provides batches

```python
from package.model import ComplexModel, SimpleModel

MODEL_REGISTRY = {
    "complex": ComplexModel,
    "simple": SimpleModel,
}

def get_model(config):
    return MODEL_REGISTRY[config.model_name](config)
```

Listing 1: Code showing the typical use of a registry

of data in the correct format to the runner. This typically involves looking up the location of a file to read from in an index, reading the relevant part of that file, performing any post-processing, and then returning the result and label as a pair.

4. **Model**. This is the core algorithm, responsible for clustering, detecting, or any other task that might be required. It is an implementation of the PyTorch `nn.Module` class.

These components are all configured using settings stored in a single global `config` object. The Hydra framework is used to create this config object from `.yaml` files. To allow changing implementations of any of these components without changing code, the code heavily uses the concept of a *registry*. This is a mapping from configuration parameters to an implementation. An example use of a registry is shown in Listing 1. In the case in that example, as in the pipeline itself, the registry is simply a dictionary.

To ease deployment of this tool, it is packaged into a docker container. The code is copied into the `/source` directory of this container and the data is mounted into the same directory. Testing is done using the `pytest` framework. There are fairly comprehensive unit tests for the dataloaders, clusterer and main training infrastructure. These helped to demonstrate correctness of the implementation. Visualisation is done via the PyTorch plugin for Tensorboard, which shows important intermediate results such as the loss and accuracy.

## 3 Data Loading

Although the job of the data loader is conceptually simple, the task of writing a data loader with good enough performance can be complex. The data loader is key to efficient training – without a fast enough data loader, the model will be waiting for batches to arrive and the GPU will be wasted. The data loader for the clusterer should be relatively simple because rumbles are very sparsely distributed throughout the dataset and so only a very small fraction of the total data needs to be considered. For the detector however, the full dataset must be loaded, which is a much more complex task. I will first discuss a naïve implementation of a data loader for the detector before discussing and analysing a few optimisations.

The data loader processes the continuous stream of data by slicing it into 10s chunks. These chunks are large enough to contain all elephant rumbles, but small enough that the rumble will not be a small component of each chunk and therefore hard to learn. A disadvantage of this approach is that chunks may contain multiple rumbles or be cut-off at the end of a chunk. However, the subsequent process of data cleaning, even if done manually, would be much less labour intensive than detection because rumbles are such a small part of the data. The data loader also has a short grace period at the start and end of the recording files to allow for any problems while setting up or removing the seismometers. On creation of the data loader, it builds an index of the relevant time chunks.

There are four different stages of optimisations that were applied to the loader.

1. **Naïve Loader**. When asked for an index, this locates the relevant slice in the file and loads it directly from the disk before computing a spectrogram using SciPy and returning the result.

2. **Cached Loader**. The cached loader tries to improve performance of the loader by caching an entire file in memory and then taking slices out of the large chunk of memory, rather than loading from disk as in the naïve loader.

3. **Pytorch Loader**. This is a reimplementation of the spectrogram generation in PyTorch instead of SciPy. This allows the the three channels to have their spectrograms computed at once, avoids conversion between tensors and numpy arrays, and allows the GPU to be used for calculating the spectra (although this does not improve performance and so is not used).

4. **Batched Loader**. This loader has a custom `collate_fn`, which is called to turn individual loaded items into a batch. This computes the spectrogram on the entire batch rather than on each item individually when they are loaded.

5. **Threaded Loader**. This loader uses 4 threads that load simultaneously to improve performance, but otherwise is identical to the PyTorch loader.

To compare performance of these loaders, I compared the time taken to load 1000 batches of size 16 on an AWS `g4dn.4xlarge` instance. The entire dataset would be 16909 batches, but this would take a prohibitively long time for some of the loaders. SqueezeNet was used as the detector to avoid performance penalties associated with a large network. The results are shown in Table 1. There are very significant performance gains for each associated optimisation, with only the batched loader not improving performance. It is also important to note that the later loaders have their true performance underestimated because the first iteration, when the files are loaded from disk, represents a significant portion of their total time. The threaded loader, for example, can complete an epoch in approximately 22 minutes, compared to the approximately 50 minutes implied by its performance here. This is a significant improvement compared to the over 100 hours that would have originally been taken by

| Loader | Time (s) | Iterations Completed |
|---|---|---|
| Naïve Loader | 2085 | 68 |
| Cached Loader | 903 | 1000 |
| Pytorch Loader | 294 | 1000 |
| Batched Loader | 314 | 1000 |
| Threaded Loader | 200 | 1000 |

Table 1: Performance comparison of different data loaders when loading seismic data

| Detector | Time for 1000 iterations (s) |
|---|---|
| ResNet | 456 |
| SqueezeNet | 200 |

Table 2: Performance comparison for different detector architectures.

the naïve loader. It is not clear why batching of spectrograms did not improve loading performance, although one possible explanation is that grouping items together results in worse cache performance when computing spectrograms than computing them individually.

## 4 Detection

The detector's purpose is to identify rumbles in the continuous stream of data. This is an important task as manual identification can be very labour intensive. Although obviously the precision and recall of a detector are important, performance is also a significant consideration as shown in Table 2, where a ResNet[1] and SqueezeNet[2] detector are compared to one another. SqueezeNet significantly outperforms ResNet, and in fact it is not really practical to train ResNet on this dataset given that each epoch takes roughly 1.5 hours compared to approximately 22 minutes for SqueezeNet (again the results in the table are affected by warm up costs).

However, although the rumbles are easily identifiable from spectrograms (see Figure 2), training the detector using a cross entropy loss function is not effective. When doing this I found that the detector simply always output there being no rumble. This is understandable when considering the class balance of the task, where 99.7% of the samples do not contain a rumble. To fix this problem, a loss such as focal loss[3] could be used, which can handle unbalanced classes much more gracefully.

## 5 Clustering

Clustering is the final part of the pipeline. It has the goal of grouping rumbles in a semantically useful way. However, as we shall see, that is not a simple result to force. The clustering method chosen was

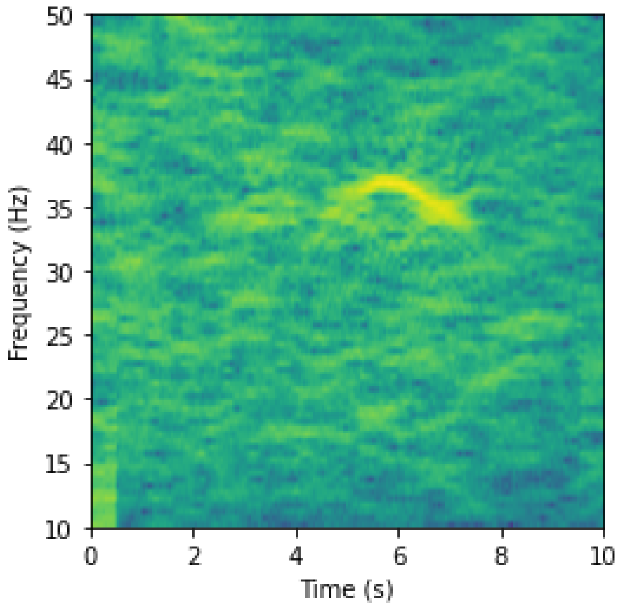

Figure 2: An example rumble from a spectrogram. The rumble is the bright n-shaped patch.

Contrastive Predictive Coding[8]. This method performs predictive coding, where future sections of the rumble are predicted, with a contrastive loss aimed at maximising mutual information between a context vector, which summarises the signal so far, and a section of the recording sufficiently far in the future. The point of this goal is that the mutual information between these sections of the signal must be some form of globally useful information because the section to predict is far in the future, and therefore no local cues (e.g. continuity) can be used to infer its structure. By preserving this when embedding into a lower-dimensional space, the algorithm must be compressing the data in a way that is somewhat meaningful. Additionally, by maximising mutual information in a latent space, there is no need for a generative model, which is computationally wasteful because effort must be put into learning local properties of the signal (such as its smoothness for example) rather than just high-level information. I will first describe some useful background behind this method before discussing its application to the elephant rumbles.

## 5.1 Contrastive Predictive Coding Background

Contrastive Predictive Coding (CPC) predicts some segment of the signal $k$ steps in the future, $x_{t+k}$, given some context $c_t$. This is done with the goal of maximising the mutual information between $x$ and $c$. To achieve this, first encode each $x_t$ into a latent space with a CNN encoder to get $z_t$, which will have a lower time resolution. Feeding the sequence $\{z_n\}_{n\leq t}$ into an auto-regressive encoder then generates $c_t$. Consider the function, $f_k$

$$f_k(x_{t+k}, c_t) = \exp\left(z_{t+k}^T W_k c_t\right) \tag{1}$$

There is a separate $f_k$ for each number of steps in the future, parameterised by the weight matrix $W_k$. We then take the sample from the future part of the signal, $x_{t+k}$, and $N-1$ negative samples from random other locations in the batch, and then compute the loss over the resulting set $X$ as

$$\mathcal{L} = -\mathbb{E}_X \left[ \log \frac{f_k(x, c_t)}{\sum_{x' \in X} f_k(x', c_t)} \right] \tag{2}$$

This is the same as the categorical cross entropy loss where the probability of predicting $x$ is $\frac{f_k(x,c_t)}{\sum_{x' \in X} f_k(x',c_t)}$. Using this, it is possible to derive that the loss maximises a lower bound on the mutual information between $x_{t+k}$ and $c_t$.

## 5.2 Implementation And Results

The implementation here uses the same parameters and architectures as in the original CPC paper[8]. That is a CNN as the encoder, which operates directly on the audio rather than transforming to a spectrogram, and a GRU as the auto-regressive encoder. A major implementation difficulty lies in the management of padding. The audio data can be of vastly different lengths, but to collect these samples into a single batch, the shorter ones must be padded. However, care must be taken to ensure that the positive and negative samples are not drawn from the padding. The negative samples are sampled randomly from the other audio in the batch. This is not an issue in the case of elephant data because a uniform-sized chunk is cut out around the rumbles. The key parameters for this model are `look_ahead` and `N`. `N` is the number of samples that are considered. I noticed that larger values of $N$ reduced the frequency with which the true positive $f_k$ had the highest score, but did not decrease the quality of the clusters and vice versa for lower values of $N$. `look_ahead` controls the maximum value of $k$ for which positive and negative samples are selected.

The clustering method applied to two datasets is shown in Figures 3 and 4. In the case of the Urban8K data, which represents very clean examples of different sounds, the clustering is easily able to separate short sounds such as gun shots and dog barks, but struggles more with longer, more droning sounds such as that of an air conditioner. One possible explanation for this lies in the choice of the `look_ahead` parameter, which controls the maximum number of steps ahead to predict from the sampled time $t$. This is a fixed number, but with variable-length signals it would probably be better to use a percentage of the total signal length. Another possible reason is poor performance of the autoregressive encoder. This could potentially be alleviated by using a more powerful autoregressive encoder such as a transformer[10]. Another explanation is that there is a length-related bug, although this code has some reasonable unit tests.

In the case of the elephant rumbles, CPC is much more successful at finding noticeable clusters, as shown in Figure 4. The elephant rumbles clustered here were identified manually, and a fixed window of size 10s cut out around each rumble. Although there are clear clusters, they are not semantically useful because they merely correspond to the different seismic stations the rumbles were taken from.

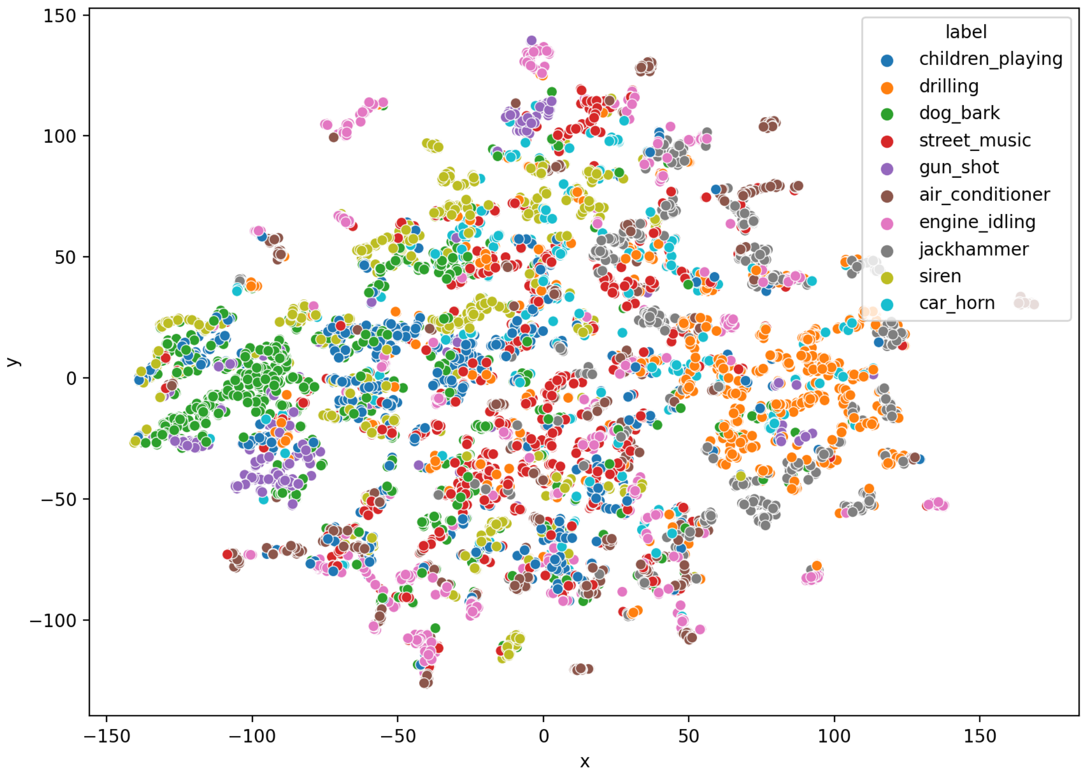

Figure 3: t-SNE[9] plot of the clusters generated on the Urban8K dataset.

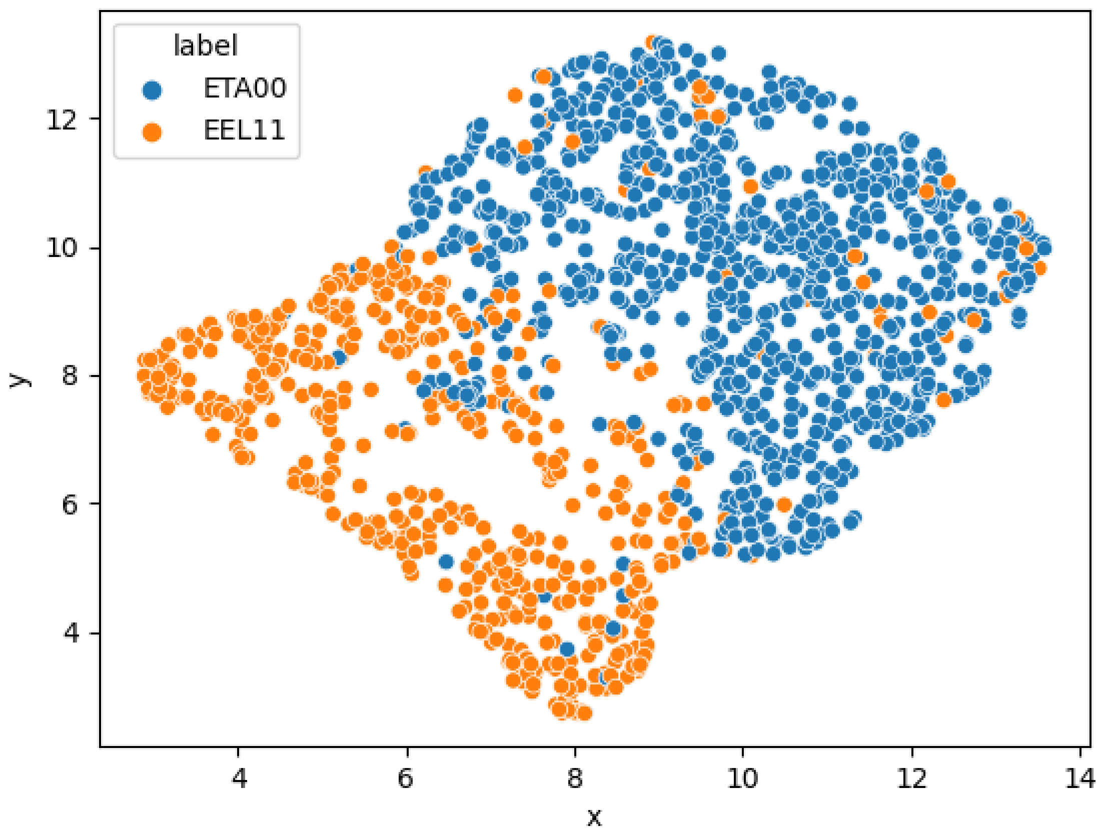

Figure 4: UMAP[4] of the clusters generated for the elephant rumbles showing their separation into the different stations

In fact, it may be the case that the clusterer does not even look at the rumble when clustering the data. This is because it is tasked only with distinguishing future samples from other chunks of audio elsewhere in the batch. If the background noise levels are different, for example, this would be sufficient to distinguish half the rumbles from one another without learning anything about the rumbles. Two possible improvements to tackle this problem would be to take positive samples from the same rumble detected at other stations, to crop the rumbles more closely in time, and to filter the frequencies of the rumbles more aggressively. The first technique, where the positive sample (i.e. the future chunk we are trying to predict) is drawn from another station's recording of the same rumble, would force the clusterer not to rely on station-specific background information. However, without sufficiently aggressive cropping of the signal in time and frequency, this is likely to result in a task that is very difficult. More aggressive filtering and cropping would be advantageous as the background would comprise a less significant proportion of the signal as a whole.

## 6 Conclusion and Future Work

Although much of the pipeline presented here is functional, much of it is not yet at a fully working stage. Therefore, the three most important pieces of future work are to apply losses that deal with unbalanced classes to the detector, to more closely crop the rumbles for clustering, and to implement a method for forcing the clusterer to cluster in a semantically useful way, for example by taking positive samples from the same rumble detected at another station. Of these, applying new losses to the detector is the most important because this will enable not only more informative clustering, but also many other types of analysis such as better location inference, where having rumbles detected at multiple stations is very important.

In this work, I presented a significant software-engineering effort towards a pipeline that can cluster elephant rumbles. However, many of the important research questions on elephant communication have sadly been left unaddressed. Despite this, I believe that the components that I have presented will serve as useful investigative tools. In particular, the loading of spectrogram data was greatly improved, and the clustering method was able to separate the rumbles in a meaningful way, but not one that presented any biological insights.

## References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[2] Forrest N. Iandola, M. Moskewicz, Khalid Ashraf, Song Han, W. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size. *ArXiv*, abs/1602.07360, 2016.

[3] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018.

[4] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018. Comment: Reference implementation available at http://github.com/lmcinnes/umap.

[5] C. E. O'Connell-Rodwell, J. D. Wood, T. C. Rodwell, S. Puria, S. R. Partan, R. Keefe, D. Shriver, B. T. Arnason, and L. A. Hart. Wild elephant (loxodonta africana) breeding herds respond to artificially transmitted seismic stimuli. *Behavioral Ecology and Sociobiology*, 59(6):842–850, Apr 2006.

[6] Katharine B. Payne, William R. Langbauer, and Elizabeth M. Thomas. Infrasonic calls of the asian elephant (elephas maximus). *Behavioral Ecology and Sociobiology*, 18(4):297–301, Feb 1986.

[7] J. Soltis. Vocal communication in African elephants (Loxodonta africana). *Zoo Biol*, 29(2):192–209, 2010.

[8] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2019.

[9] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.