

Term Research

Finding Anomalies in Large Unlabeled Acoustic Data

Philip Robinson

Oregon Health Sciences University

December 15, 2019

Abstract

University of Hawaii’s Aloha Cabled Observatory (ACO) has approximately 10 years of unlabeled hydrophone data, in need of indexing, labeling, and cleaning; to more easily extract acoustic anomalies. Currently searching the audio recordings is done at human pace by listening to the audio stream, or looking at it’s spectrogram, which takes ~ 2 -20 minutes for every 5 minutes of audio. This is intractable at scale. Unfortunately, audio data is known to be high dimensional, making it very difficult to label raw input streams. Additionally, the properties of acoustic anomalies are not necessarily known a priori. It is known that unsupervised anomaly detectors, leveraging Variational Auto-Encoders are successful for acoustic anomaly detection [2]. This is an exploration of one of these models as a means to identify when whales are vocalizing.

1 Introduction

Scientists, with University of Hawaii’s Aloha Cabled Observatory (ACO), have gathered 10+ years of continuous audio for acoustic ocean survey studies. Currently searching the audio recordings is done at human pace by listening to the audio stream, which takes ~ 2 -20 minutes for every 5 minutes of audio, depending on experience levels. Even at its fastest, this is intractable at scale. Often ocean hydrophone observatories are used to survey and track cetations vocalizations for population measurement, acoustic analysis for earthquake/tectonic events, and applications in monitoring vessel traffic and activities. The ACO hydrophone data is unlabeled and growing at 1.2 terabytes per year.

“Being able to automatically detect whale calls in an un-monitored system can help identify time-series of whale locality to the area. Being able to apply this detection algorithm to a long time-series of sub-sea audio, such as that from the ACO, allows scientists to derive a time-series of whale activity in the area. This can help us study the relationship between migration patterns and known climate events.”

- Kellen Rosburg
Senior Ocean Computer Specialist
OOI Cabled Array, APL/UW

In the most general form, it would be extremely useful to have a generic unsupervised acoustic anomaly detector, for hydrophone data. This would allow a human or machine reviewer to focus on only interesting data in the task of event sorting and classification.

2 Model

In prior work on acoustic anomaly detection [1] address the generic task of anomaly detection, over image encoded acoustic anomalies. A Variational Auto-Encoders is expected to behave as an identity function, however reconstruction error can be an indicator of content unseen in the training data. When a Variational Auto-Encoders is trained on typical data, a threshold over an anomaly score can be used for to flag atypical data. The paper proposes a minimization of false positive rate, given a constrained true positive rate, by expressing a threshold of reconstruction error.

Proposed is an anomaly generating network. This is done by fitting the Variational Auto-Encoders’s gaussian manifold on general data, generating an under-specified probability density function (by minimizing KL divergence), then tighter fitting a more descriptive probability density function over the latent vectors of typical data projected into the

gaussian manifold. Once both distributions are fit, rejection sampling in the form of sampling from the general distribution values of low probability from the descriptive distribution.

Training is broken up into three major phases. These phases contribute to fit the generator and discriminator networks, as well as a typical data model (in this case our GMM) for informing rejection sampling.

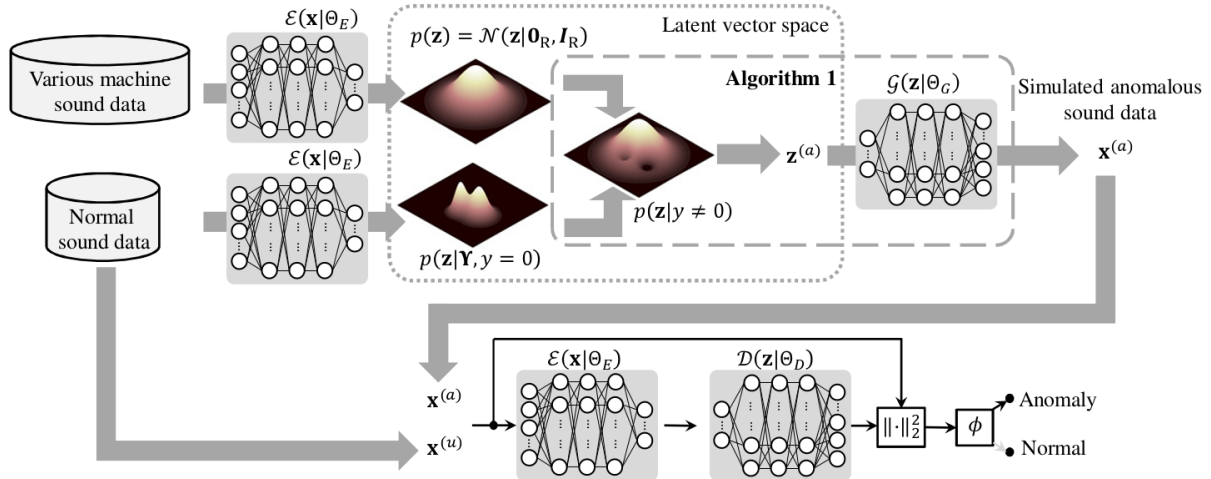
The first phase updates the parameters of the encoder and the generator for all input data. The goal of this process is to define the general probability density function of the latent space of all data and a mapping to a unit gaussian manifold.

The second phase updates the parameters of the **encoder** and **discriminator** with a loss function that is defined by reconstruction error against a thresh-

old, and an approximation of true/false positive rates (which cannot be calculated directly). This phase requires sampling acoustic anomalies from the unit gaussian and rejecting likely events measured by the specific probability density function, through the generator network. For the ACO typical data can be sampled from the evenings or low traffic months, as cetations tend to not be as vocal in the evenings; this strategy is also resilient to accumulative damage/drift of the equipment over time.

The third phase is to update the specific probability density function modeled by a GMM. This is repeated as necessary.

Discrimination is accomplished by learning an appropriate threshold for the likelihood of a latent representation of an event.



3 Topics

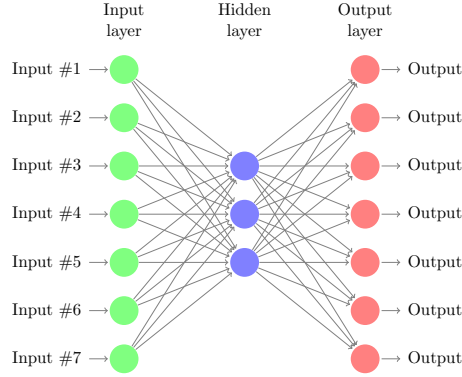
Due to the complex nature of this architecture, it is important to provide an understanding and background of supporting topics. This section introduces the pre-requisite knowledge to understand the final model.

3.1 Auto-Encoders

Auto-Encoders are an unsupervised neural network model that attempts to constrain an identity function. This is usually done by optimizing the weights of the neural network to reconstruction error, while passing through a low dimensional **bottle**.

$$J^R = ||Input - Output||_2^2$$

The **bottle**, represented as a narrowing of nodes in the neural network, expresses a compressed representation of the original content wrt it's neighboring encoding and decoding networks. These networks can be thought of as learning a lossy compression and decompression functions. The lossy characteristic of these mapping functions is expressed in image data as a blurry recreation of the original content.



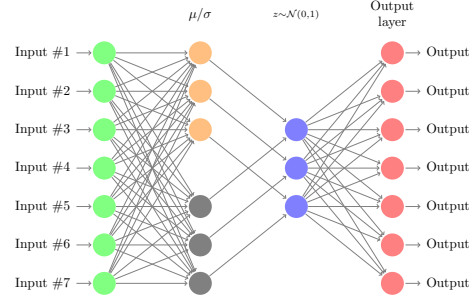
../ae.py

```

146 class AutoEncoder(BottleNetwork):
147     _reconstruction_error = partial(F.mse_loss, size_average=False)
148     # _reconstruction_error = partial(
149     #     F.binary_cross_entropy, size_average=False
150     # )
151
152     def __init__(
153         self, *, bottle_size, data_shape, EncoderType, DecoderType,
154         encoder=None
155     ):
156         super().__init__(bottle_size=bottle_size, data_shape=data_shape)
157         self._encoder = encoder if encoder \
158             else EncoderType(bottle_size=bottle_size, data_shape=data_shape)
159
160         self._decoder = DecoderType(
161             bottle_size=bottle_size, data_shape=data_shape
162         )
163
164     def parameters(self):
165         return chain(self._encoder.parameters(), self._decoder.parameters())
166
167     def encode(self, X):
168         return self._encoder(X)
169
170     def decode(self, z):
171         return self._decoder(z)
172
173     def forward(self, X):
174         h = self.encode(X)
175         Y = self.decode(h)
176         return Y
177
178     @classmethod
179     def reconstruction_error(cls, Y, X):
180         return cls._reconstruction_error(Y, X)
181
182     @classmethod
183     def loss(cls, X, *args):
184         Y, *_ = args
185         return cls.reconstruction_error(Y, X)

```

$$J^{KLR} = J^R + KL(z || \mathcal{N}(0, 1))$$



../vae.py

```

23 class VariationalAutoEncoder(AutoEncoder):
24     def __init__(
25         self, *, h_size, z_size, data_shape,
26         EncoderType, DecoderType, encoder=None
27     ):
28         super().__init__(
29             bottle_size=h_size, data_shape=data_shape,
30             EncoderType=EncoderType, DecoderType=DecoderType,
31             encoder=encoder
32         )
33         self.z_size = z_size
34         self.h_size = h_size
35
36         self.mu = nn.Linear(h_size, z_size)
37         self.logsigma = nn.Linear(h_size, z_size)
38         self.eta = nn.Linear(z_size, h_size)
39
40     def parameters(self):
41         return chain(
42             super().parameters(),
43             self.logsigma.parameters(),
44             self.eta.parameters(),
45             self.mu.parameters()
46         )
47
48     @classmethod
49     def _reparameterize(cls, mu, logsigma):
50         std = logsigma.mul(0.5).exp_()
51         # return torch.normal(mu, std)
52         epsilon = torch.randn_like(mu)
53         z = mu + std * epsilon
54         return z
55
56     def bottle(self, h):
57         mu = self.mu(h)
58         logsigma = self.logsigma(h)
59         z = self._reparameterize(mu, logsigma)
60         return z, mu, logsigma
61
62     def decode(self, z):
63         h = self.eta(z)
64         return self._decoder(h)
65
66     def _sample(self, n):
67         z = torch.stack([GaussianSample(self.z_size)
68             for _ in range(n)])
69         return z
70
71     def generate(self, n=1):
72         z = self._sample(n)
73         with torch.no_grad():
74             h = self.eta(z)
75             return self._decoder(h)
76
77     def forward(self, X):
78         h = self.encode(X)
79         z, mu, logsigma = self.bottle(h)
80         Y = self.decode(z)
81         return Y, mu, logsigma
82
83     @classmethod
84     def _KL_loss(cls, mu, logsigma):
85         return -0.5 * torch.mean(
86             1 + logsigma - mu.pow(2) - logsigma.exp()
87         )
88
89     @classmethod
90     def _vae_loss(cls, X, *args):
91         Y, mu, logsigma, *_ = args
92         return cls._KL_loss(mu, logsigma) + \

```

3.2 Variational Auto-Encoders

Variational Auto-Encoders are similar to Auto-Encoders, however they restrict the form of manifold forming the **bottle** to a gaussian manifold. It is important to note that trained Variational Auto-Encoders can be split at the **bottle**, and used like a generative model. This is done by sampling from a unit gaussian; the samples is interpreted as a latent representation of an input from the **encoder**. The **decoder** is then used on this sample to generate new content that would have been encoded as the latent representation.

In order to train this style of network, the loss function is complimented by Kullback Leibner divergence, to constrain the latent space to a gaussian manifold.

3.3 Rejection Sampling

Rejection sampling is a technique used in many statistical models. Many distributions are difficult to

sample from, however simple to evaluate likelihood against. A second distribution whose domain is unbiased wrt the original distribution may be elected to act as a proxy for sampling.

The simplest example of this is in attempting to sample points from a unit circle. There exist known algorithms to sample from a uniform distribution. A two dimensional uniform distribution is equivalent to sampling from a rectangle. Rejection sampling can be used to sample from the unit circle by electing a 2 dimensional uniform distribution that encompasses the area of the target circle, and rejects points under the constraint that $x^2 + y^2 \leq 1$.

This relates to Variational Auto-Encoders because there exists known algorithms for sampling from gaussian distributions. We can elect any statistical model to describe typical data, that shares an unbiased domain with a gaussian. This typical distribution can then be used to form a likelihood threshold for rejection criteria. This strategy informs the core of the proposed model, further explained by the Neyman-Pearson Lemma.

3.4 Neyman-Pearson Lemma

The Neyman-Pearson Lemma states that the likelihood ratio is the ‘uniformly most powerful discriminator’ for a statistical hypothesis test. This is leveraged into forming a loss function by increasing the reconstruction error of anomalous data, and decreasing the reconstruction error of typical events in data.

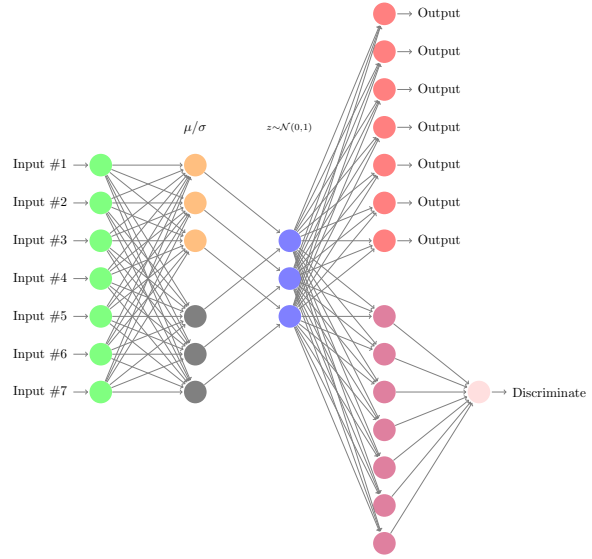
$$J^{NP} = FP - TP$$

Theoretically a discriminator network leveraging this loss function, with known anomalous and typical data, will become better identity function for typical data and increasingly poor at reconstructing atypical data.

4 Gumiho Networks

Although this term isn’t in the common nomenclature, a gumiho network refers to a network that has more than one tail, generating output. The intent is to use different loss functions depending on which network output is triggered. For Auto-Encoders and Variational Auto-Encoders this allows for the learned encoding function to generate the same manifold, while satisfying very different

loss enforced constraints. It is helpful, although not necessary, to choose non-competing loss functions.



Since the discriminator network uses a threshold over reconstruction error, the Variational Auto-Encoders (that uses only recognition error) is not considered a non-competing loss function. This conditional training, dependent on the evaluation path lends itself to `pytorch` [3] over `keras`. This will be discussed in greater detail in the next section.

5 Implementation

Systems like `keras` and `tensorflow` use a precompiled execution graph, to facilitate more predictable back propagation. For this model, there is a parameterized discriminating function in the `bottle` of the network. Additional complexity is added, due to the sharing `encoder` variables across multiple `decoder` networks. `pytorch` [3] doesn’t restrict the model to precompiled execution graphs and allows for use of the `torch.no_grad()` context manager in order to restrict when variables can be treated to not need back propagation.

The user provides an `encoder`, `decoder`, the interfacing dimension size h , the `bottle` dimension size z , and the count of mixtures for the GMM M . The `encoder` and `decoder` are expected to reduce the input to and from a linear layer of size h . A linear `bottle` layer is provided to map from h to z dimensions and extract μ and σ to inform KL loss by reparameterization. The `decoder` is then preceded by a provided layer η that maps back from z to h dimensions. This abstraction layer makes it

much easier to modularize the network.

The code represents a direct inheritance principled on object oriented design of machine learning models. Discriminator Network is a Conditional Generating Network, is a Gumiho Network, is a Variational Auto-Encoders , is a Auto-Encoders.

5.1 Gumiho Network

The Gumiho Network is the first unfamilure network abstraction, and therefore starts the descriptive sections of this text.

The Gumiho Network is a Variational Auto-Encoders with multiple goal states. A user can add additional decoders and cooresponding loss functions. This allows a trained model to develop an embedding optimized, not only for reconstruction but, for various additional goals like classification. The `add_tail(self, network, loss)` method requires the network can be fed by an h dimensional linear layer.

../gumiho.py

```

14 class GumihoNetwork(VariationalAutoEncoder):
15     def __init__(self, *args, **kwargs):
16         super().__init__(*args, **kwargs)
17         self.tails = {}
18         self.losses = {}
19         self.add_tail(None, self._decoder, self._vae_loss)
20
21     def parameters(self):
22         tails_params = (
23             self.tails[key].parameters()
24             for key in self.tails
25             if key is not None # because it is already added
26         )
27         return chain(super().parameters(), *tails_params)
28
29     def forward(self, X, *, tail=None):
30         h = self.encode(X)
31         z, mu, logsigma = self.bottle(h)
32         Y = self.decode(z, tail=tail)
33         return Y, mu, logsigma
34
35     def decode(self, z, *, tail=None):
36         if tail is None: # XXX make advanced context manager
37             h = self.eta(z)
38         else:
39             with torch.no_grad():
40                 h = self.eta(z)
41         return self.tails[tail](h)
42
43     def _generate_from_z(self, z, tail=None):
44         with torch.no_grad():
45             return self.decode(z, tail=tail)
46
47     def generate(self, n, *, tail=None):
48         z = self._sample(n)
49         return self._generate_from_z(z, tail)
50
51     def add_tail(self, key, network, loss):
52         self.tails[key] = network
53         self.losses[key] = loss
54
55     def loss(self, X, *params, tail=None):
56         return self.losses[tail](X, *params)

```

5.2 Conditional Generating Network

In a Variational Auto-Encoders , the embedding is mapped to a gaussian manifold. This means that sampling from a unit gaussian and passing

data through a decoder will generate results cooresponding to the decoder's goal. When used with the tail associated with the reconstruction task, this produces viable input data. the Conditional Generating Network allows condition to be provided to the sampled value and the tail elected to be specified by the user. For the final model this is a mechanism to allow the GMM to be used for conditioning and the reconstructor to be used to generate anamolies as found in the reference materials.

../gumiho.discriminator.py

```

37 class CondGeneratorNetwork(GumihoNetwork):
38     def __init__(self, *args, **kwargs):
39         super().__init__(*args, **kwargs)
40         self.conds = {}
41         self.add_cond(None, self._none_cond)
42
43     @classmethod
44     def _none_cond(cls, x):
45         return True
46
47     def _sample(self, n, *, tail=None):
48         def _local_gen():
49             while True:
50                 z = GaussianSample(self.z_size)
51                 if self.conds[tail](z):
52                     yield z
53
54         g = islice(_local_gen(), n)
55         f = [identity.remote(_) for _ in g]
56         z = ray.get(f)
57         Z = torch.stack(z)
58         return Z
59
60     def add_cond(self, key, cond):
61         self.conds[key] = cond
62
63     def generate(self, n, *, cond=None, tail=None):
64         z = self._sample(n, tail=cond)
65         return self._generate_from_z(z, tail=tail)

```

5.3 Discriminator Network

The Discriminator Network models the machine specified [1] for unsupervised anomaly detection. There are three tails, one for training the GMM, one for discrimination, and one for reconstruction.

../gmm.py

```

37 class GMM(nn.Module, TOPROB):
38     def __init__(self, count, dims):
39         super().__init__()
40         self.count = count
41         concentration = torch.Tensor([1/count] * count)
42         Dir = torch.distributions.dirichlet.Dirichlet(concentration)
43         self.mixtures = nn.ModuleList(
44             [_Mixture(dims, phi=phi) for phi in Dir.sample()]
45         )
46
47     @property
48     def epsilon(self):
49         return (torch.rand(1) - 0.5) * EPSILON
50
51     def mixed_nll(self, X):
52         ll = self(X)
53         return -ll.sum(axis=1)
54
55     def forward(self, X):
56         ll = t(torch.stack([model(X) for model in self.mixtures])).squeeze()
57         return ll
58
59     def update(self, X, log_affiliations, show=False):
60         prob = self.norm_prob(self.aff_to_prob(log_affiliations))
61         for idx, model in enumerate(self.mixtures):
62             p = prob[:, idx].unsqueeze(1)
63             if not model.update(X, p):
64                 raise "hell"
65
66     if show:
67         _phi = torch.Tensor([model.phi for model in self.mixtures])
68         print('summary:', prob.argmax(dim=0))
69         print('phi:', _phi)

```

The code for this model is found in the repository instead of this paper due to its complexity.

6 Data

The model is build to be agnostic to types of **encoder** and **decoder**. The expectation of the user is to provide models appropriate for their data. It is also expected that a loading **coroutine** is provided. The coroutine needs to take in the count of a batch in order to be trained in a consistant way.

6.1 MNIST

The MNIST data hopes to example hand written single digit entries. The original intent is to inform algorithms of had written digit recognition for the postal service. For this term project, MNIST was used to more consistently and reproducibly explore model building.

../ae.py

```
188 def data_initializer(*,
189     censored=[2, 3, 4, 5],
190     atypical=[1, 7],
191     various=0,
192     data_shape
193 ):
194     HERE = Path(".").
195     _ = torch.utils.data.DataLoader(
196         datasets.MNIST(
197             HERE,
198             train=True,
199             download=True,
200             transform=transforms.Compose([
201                 transforms.ToTensor(),
202                 transforms.Normalize((0.0,), (1.0,))
203             ]),
204             shuffle=True
205         )
206
207     typical_stream = cycle(
208         x.squeeze()
209         for x, y in _
210         if not (y in censored or y in atypical)
211     )
212
213     def to_batch(iterable):
214         while True:
215             n = (yield)
216             batch = torch.stack(
217                 [x.view(data_shape) for x in islice(iterable, n)]
218             )
219             yield batch
220
221     if not various:
222         return to_batch(typical_stream)
223     else:
224         various_stream = cycle(
225             x.squeeze()
226             for x, y in _
227             if (y in censored)
228             or (y in atypical and not randint(0, various))
229             or (y not in censored and y not in atypical)
230         )
231     return to_batch(typical_stream), to_batch(various_stream)
```

In this trained model, the digits listed as {1,7,4} were labeled as anomalous. Any other digit were described as typical events. The expectation of a trained network is to generate high accuracy reconstructions of {2,3,5,6,8,9} and low accuracy reconstructions of {1,7,4} compile.



The images above are sources and their paired results from passing the MNIST data through the trained Variational Auto-Encoders with loss of mean squared error. It is apparent by adjusting the networks that using convolutions and ending the networks with sigmoid result in much higher quality yields.

6.2 Aloha Cabled Observatory

The ACO data is gathered at N22°45.110' W158°00'. The recordings are encoded as, custom, variable bit-width raw peak sensor readings. Each file is saved as a 5 minute duration (barring hardware related problems), named with it's datetime stamp. The set elected for this project was 44100 samples per second. The developed **ACDio.py** library allows simple manual indexing and datetime-search of the data.

```
1 from aco import ACDio, datetime, timedelta
2
3 loader = ACDio('./basedir/')
4 target = datetime(
5     day=18, month=2, year=2016,
6     hour=7, minute=55
7 )
8
9 src = loader.load(datetime)
10 snip = src[
11     timedelta(seconds=7):
12     timedelta(seconds=11)
13 ]
14 snip.View()
```

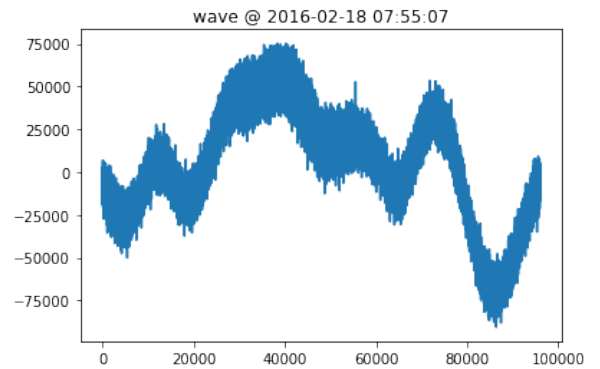


Figure 1: Raw Data

It is visible, from (Figure 1) that the direct current gain is not centered at zero, nor trivially accumulative. This is a consequence of changes in atmospheric pressure, due to the ocean's motion, effecting the signal.

It is also not obvious this track has a vocalization, highlighted in (Figure ??). This pattern is indicative of high amounts of noise, and is expected for all samples.

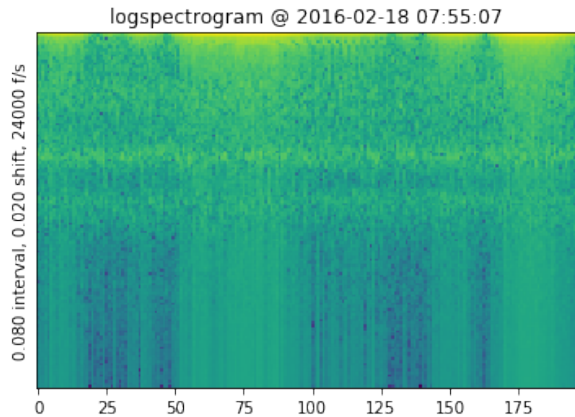


Figure 2: Raw Spectrogram

Inspired by these plots, and studies in signal processing, the audio track can be represented as an image. Expressing acoustic signals as spectrogram and mel-frequency cepstrum algorithms, is a common way to enable these models. This representation lends itself to many deep learning models. Additionally, this allows reasonable evaluation of a target model by using any well studied image dataset. For this project the target dataset is described in the next subsection.

7 Results

With a small enough dataset and low enough embedding dimension, the model trains. It is observable that training decreases loss making true positive rate and false positive rate separate, respecting the expected behaviour by the Neyman-Pearson Lemma. It is also clear that changing out the encoder and decoder networks between linear and convolutional significantly improves performance for reconstructions. Finally, tailing the decoder network with a sigmoid layer improves performance as well. As an alternative to using sigmoid finishing layer, the model can be trained on binary cross entropy loss, but in conjunction seems to have no better performance.

The GMM has underflow errors when appropriate dimensions for the ACO dataset are selected. To treat this the forward pass generates log-likelihoods,

however there are complications in the update step that prevent continuing to work in log space, once we add the ϕ_i parameters.

../gmm.py

```

71 class Gaussian(nn.Module, TOPROB):
72     def __init__(self, dims):
73         """
74         networks will not require gradients, because we will not use
75         '.update()' to propagate updates
76         """
77         super().__init__()
78         self.dims = dims
79
80         self.mu = (
81             nn.Parameter(
82                 torch.rand(dims), requires_grad=False) - 0.5) * 2.0
83         self.sigma = nn.Parameter(
84             torch.eye(dims),
85             requires_grad=False
86         )
87
88     @property
89     def epsilon(self):
90         return (
91             torch.rand(
92                 (self.dims, self.dims)
93             ) - 0.5) * EPSILON
94
95     def nll(self, X):
96         return -self(X)
97
98     def prob(self, X):
99         return self.aff_to_prob(self(X))
100
101     def forward(self, X):
102         mm = partial(reduce, torch.mm)
103
104         with torch.no_grad():
105             dims = torch.Tensor([self.dims])
106             log_two_pi_dims = dims * (
107                 torch.log(torch.Tensor([2.0]))
108                 + torch.log(torch.Tensor([math.pi]))
109             )
110
111             sigma = self.sigma + self.epsilon
112             inv_sigma = torch.pinv(sigma)
113             log_det_sigma = sigma.logdet()
114
115             log_sqrt_den = 0.5 * (log_two_pi_dims + log_det_sigma)
116
117             collect = []
118             for x in X:
119                 diff = (x - self.mu).unsqueeze(0)
120                 log_exp_num = mm([-0.5 * diff, inv_sigma, diff.t()])
121                 ll = log_exp_num - log_sqrt_den
122                 collect.append(ll)
123             out = torch.cat(collect)
124             return out
125
126     def _update_m(self, X, affiliations):
127         return (X * affiliations).sum(axis=0) / affiliations.sum()
128
129     def _update_s(self, X, _m, affiliations):
130         diff = (X - _m)
131         _acc = []
132         for s, g in zip(diff, affiliations):
133             _ = s.unsqueeze(0) # make transposable
134             _ = g * torch.mm(_t(), _)
135             _acc.append(_)
136         num = torch.stack(_acc).sum(axis=0)
137         den = affiliations.sum()
138         _s = num / den
139         return _s
140
141     def _update(self, X, affiliations):
142         """
143         probs is the probability of x_i being represented by the current
144         Gaussian model.
145         """
146         if affiliations is None:
147             count = X.size(0)
148             affiliations = torch.ones((count, 1))
149         _m = self._update_m(X, affiliations)
150         _s = self._update_s(X, _m, affiliations)
151         return _m, _s
152
153     @classmethod
154     def _isdumb(cls, tensor):
155         return (torch.isnan(tensor) + torch.isinf(tensor)).sum()
156
157     def update(self, X, affiliations=None):
158         with torch.no_grad():
159             _m, _s = self._update(X, affiliations)
160             if self._isdumb(_m) + self._isdumb(_s):
161                 return False
162
163             self.mu.data = _m
164             self.sigma.data = _s
165             return True

```

¹<https://github.com/probinso/gumiho-network>

The code can be found in it's current state on github.¹

References

- [1] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Yuta Kawachi, and Noboru Harada. Unsupervised detection of anomalous sound based on deep learning and the neyman-pearson lemma. 2018.
- [2] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Yuta Kawachi, and Noboru Harada. Unsupervised detection of anomalous sound based on deep learning and the neyman-pearson lemma. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 27(1):212–224, January 2019.
- [3] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.