THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

A Partitioned Autoencoder for Audio De-Noising

by

Ethan Lusterman

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

September 2016

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE
ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

_____

Dean, School of Engineering   Date

_____

Prof. Sam Keene, Thesis Advisor   Date

Acknowledgements

This thesis would not be possible without the guidance and support from my advisor, Dr. Sam Keene. He has mentored me since I was an undergraduate, and I am grateful for him helping this project come to life. I also want to thank Christopher Curro, my informal second advisor who helped me to think outside the box and for whom the overall system architecture is named after.

I would like to thank Kate Thorsen for pushing me past my potential and encouraging me to stay positive despite the frustrations of research. Lastly, I would like to thank my friends and family for their support. This thesis would not have been possible without all their support.

Abstract

Traditional audio denoising systems are often linear time-invariant (LTI) and often require access to clean data to properly train to remove noise. Since clean audio is often unavailable, we build on a partitioned denoising autoencoder for denoising audio signals when clean examples are unavailable for training. In addition, the nonlinearity of a neural network architecture provides additional gains over standard linear models. We compare existing semi-supervised denoising systems as well as canonical supervised denoising autoencoders. We show that for moderate levels of noise, our autoencoder can outperform existing schemes.

Contents

List of Figures

Table of Nomenclature

1 Introduction

Advances in smartphone technology have led to smaller devices with more powerful audio hardware, allowing for common consumers to make higher quality recordings. However, recorded speech and music are subject to noisy conditions, often hampering intelligibility and listenability. The goal of denoising audio recordings is to improve intelligibility and perceived quality. A variety of applications of audio denoising exist, including listening to a recording of a band or an artist's live performance in a noisy crowd, or listening to a recorded conversation or speech under noisy conditions.

A common technique for denoising involves the use of autoencoder neural networks. [2] Advances in parallel graphics processing units (GPU) and in machine learning algorithms have allowed for training deeper networks faster, utilizing more hidden layers with more neurons.

Prior work in denoising audio has involved the use of noise-free training data. Since common consumers do not often have access to clean audio, we seek to denoise without the use of clean audio. Other work has touched on such a semi-supervised scenario but was used more as a preprocessing step to a classification algorithm than as time-domain denoising. [1]

In this thesis, we compare several neural network architectures and problem scenarios, ranging from data input types, level of noise, depth of network, training objectives, and more. In Chapter 2, we present background information on machine learning, neural networks, and signal processing as well as prior work in audio denoising. In Chapter 3, we detail the problem formally as well as introduce our signal model and sourced data. In Chapter 4, we detail all considered network architectures. In Chapter 5, we compare results

from different data inputs, levels of noise, network architectures, and training objectives and discuss methods of evaluation. Finally, we make conclusions and recommendations for future work in Chapter 6.

2 Background

2.1 Machine Learning

Machine learning involves the use of computer algorithms to make decisions based on training data. Generally, this falls into categorizing input data (classification) or determining a mathetmatical function to determine a continuous output given an input (regression). Popular classification examples include recognizing handwritten digits (MNIST) as well as determining whether an image contains a cat or a dog. (REF) An example of a regression problem is determining the temperature given a set of input features (humidity, latitude, longitude, date, etc.).

Problems where training data contain input data vectors as well as the correct output vectors (targets) are known as supervised learning problems. Training a model to denoise audio where noise was introduced to the clean audio would be a supervised learning problem. On the other hand, training a model to denoise audio where the underlying clean signal is not known is an unsupervised learning problem. Different loss (objective) functions and neural network architectures can be exploited to accomplish denoising without the clean data.

For the purposes of this thesis, we use machine learning to determine an underlying nonlinear function that removes noise from time slices of audio (i.e. regression). These slices can then be pieced back together through overlap-add resynthesis. To clarify, this is a general linear model that maps an input noisy audio vector $y[n] = x[n] + N[n]$ to $\tilde{x}[n]$, a target denoised audio vector, where $x[n]$ is the underlying clean signal and $N[n]$ is the additive background noise.

2.1.1 Regression

A classical regression technique is linear regression, where one or more independent variables x_i are used to determine a scalar dependent variable y. The case of a single independent variable x is known as simple linear regression. More formally, for k independent variables, we would like to determine a weight vector $\mathbf{w}$ and bias vector $\mathbf{b}$:

$$y_i = w_1 x_{i1} + \cdots + w_k x_{ik} + b_i, \qquad i = 1 \ldots, n \tag{1}$$

$$\mathbf{y} = \mathbf{x^T w} + \mathbf{b} \tag{2}$$

where the rows of x^T are the example input observations and $\mathbf{y}$ and $\mathbf{b}$ are column vectors.

By extension, the case of linearly estimating a vector output giving a vector input is known as a generalized linear model. A canonical example would be estimating a sine wave $x[n]$ over some number of N samples given noisy samples $y[n] = x[n] + N[n]$.

2.2 Neural Networks

In this thesis, we deal only with feed-forward neural networks, which are essentially directed acyclic graphs (DAG) for computation. In other words, information only moves through the network in one direction. An example neural network is shown in Figure 1.

The connections in a nerual network can be represented by linear combinations of the input variables with learned weights $\mathbf{w}$. [3] Unlike standard linear models however, neural networks apply a nonlinear activation $f(.)$ at the output of each neuron. The circle nodes in a neural network diagram can be thought
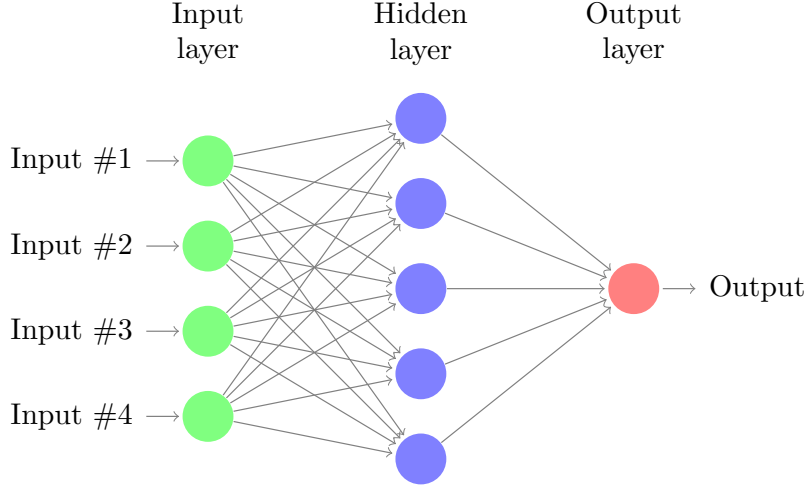
Figure 1: An example neural network. There are 4 input variables, 1 hidden layer with 5 neurons, and 1 output variable.

of as the sum of the linear combinations of the connection edges and the application of the bias and activation function. Therefore, a hidden neuron z_j in a network with N input variable nodes, M hidden nodes, and K output nodes takes on the value

$$z_j = f(a_j) \tag{3}$$

where the activiation a_j is given by

$$a_j = \sum_{i=1}^{N} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{4}$$

The connection values w_{ji} are referred to as weights, and the scalars w_{j0} are referred to as biases. Note that the superscripted numbers refer to the Then, the output y_k is given by

$$y_k = g(a_k) \tag{5}$$

where the output activation ak is given by

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{6}$$

We are free to choose activation functions, which we will discuss later. However, note that at the output, the function $g(\textbf{.})$ is often an identity for regression problems and a sigmoid $\sigma(\textbf{.})$ for classification problems.

Often, the weights and biases are grouped into a weight vector $\mathbf{w}$. In other words, similar to the linear models described earlier, a neural network is a nonlinear function of input variables $\{x_i\}$ to output variables $\{y_k\}$ where the parameters of the function are learned via training techniques.

2.2.1 Dense Layer

Described in the previous section, we refer to a dense layer as a fully connected neural network, in which no interconnections between neurons are missing at each layer. Dense layers can be prone to overfitting. However, as we mention later, overfitting is not an immediate concern for the purposes of this thesis.

2.2.2 Denoising Autoencoder

An autoencoder is normally an abstraction of neural networks in which an encode function $\mathbf{Z} = f(\mathbf{X})$ and a decode function $\hat{\mathbf{X}} = g(\mathbf{Z})$ are learned to learn a lower-dimensional representation of some input $\mathbf{X}$. [1] A denoising autoencoder is a supervised process whereby the clean input is first corrupted by some stochastic process $\mathbf{Y} = u(\mathbf{X})$. In other words, the neural network input would be a noisy input Y, and the network would try to learn weights such that the network output $\widehat{\mathbf{X}}$ approximates the clean input $\mathbf{X}$. Another way to frame it is that your network is learning the inverse function of the noise process $u(\mathbf{x})$.

2.2.3 Network Training

In order to train a neural network, we must update the weights such that we minimize a loss function, often some kind of sum-of-square error function. [3] Often, a stochastic gradient descent (SGD) approach is taken to determine the weights that minimize the loss function.

2.2.4 Choice of Activation Function

The most common activation functions used are the logistic sigmoid function and the hyperbolic tangent. [2] The logistic sigmoid function is given by

$$g(x) = \frac{1}{1 + \exp(-x)} \tag{7}$$

Note that the sigmoid function has an output on the range $(0, 1)$. The hyperbolic tangent function (tanh) is given by

$$g(x) = \frac{\sinh x}{\cosh x} \tag{8}$$
$$= \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{9}$$
$$= \frac{1 - \exp(-2x)}{1 + \exp(-2x)} \tag{10}$$

Note that the hyperbolic tangent function has an output on the range $(-1, 1)$.

Generally, our choice of nonlinearity should be chosen such that the expected range of desired output matches the nonlinearity's. In the case of audio denoising, different activations can be chosen depending on the input format. For example, time-domain audio frames are often processed with a digital floating point representation on the range of $[-1, 1]$. In such a case, the hyperbolic tangent might be appropriate. On the other hand, if we were working with

magnitude spectra of an audio signal, we would use a linearity with an output range of $[0, \infty]$.

Recently, a more popular activation function which has in use is the rectified linear unit (ReLU). [4] The ReLU is defined by the following:

$$
g(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{11}
$$

In other words, $g(x) = \max(0, x)$. This function satisfies the range of output we expect for magnitude spectra. In terms of gradient calculations, the zero derivative for negative input values of x can cause nodes to not be activated, potentially leading to gaps in information at the output and slower training time. To combat this, variations of the ReLU are used which have small, non-zero gradients for negative input values. For example, leaky ReLU's are defined by

$$
g(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{12}
$$

Advantages of ReLU's include better gradient propagation as well as fast computation and sparse representation. Some disadvantages include non-differentiability at $x = 0$. Also, depending on use case, sparse representation might not be desired.

2.2.5 Minibatch Training

Historically, neural networks were trained one example at a time (online) or in a batch (all examples at once). [5] For the online approach, the network weights are updated after gradients are calculated and backpropagated for

each training example. On the other hand, the batch approach accumulates average gradients for all examples and then updates the network weights. The batch approach might approximate the true gradients better than the online approach, but the online approach tends to have faster training time and convergence. [5] This is because with an online approach, the network is less likely to get stuck in a local minimum.

Minibatch training has become more popular recently. Serving as a midway point between the two approaches, minibatch training exposes the network to a small number of examples and then accumulates gradients and updates the network weights. The trend toward minibatch training comes at a time where parallel computing resources are easily accessible.

2.2.6 Batch Normalization

Batch normalization is a technique that helps to speed up training time and convergence. Batch normalization accumulates learned statistics of the network to help achieve loss convergence more quickly. More formally, an input minibatch x is normalized by the following:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{13}$$

[6]. During training, the minibatches are normalized to zero-mean, unit-variance and transformed by parameters γ and β. At inference time, the learned parameters are instead used, which are made up of the average statistics from training.

Batch normalization prevents activations from saturating from widely varying input minibatches. This allows us to use faster learning rates and be less careful about how to initialize our parameters. [6]

9

2.3 Signals and Systems

Domain knowledge of discrete audio signals and systems better informs our decisions for an audio denoising system, so some background information on signals and systems as it pertains to this thesis is detailed below.

2.3.1 Signals

We deal exclusively with discrete-time audio signals in this thesis. A discrete-time audio signal $x[n]$ is represented as a sequence of numbers (samples), where each integer-valued slot n in the sequence corresponds to a unit of time based on the sampling frequency f_s. This comes from sampling the continuous-time audio signal $x_c(t)$:

$$x[n] = x_c(nT) \tag{14}$$

where $T = 1/f_s$. For example, a 1-second speech signal sampled at 8kHz has 8000 samples. Furthermore, digital signals also have discrete valued sample amplitudes. For the purposes of this thesis, the bit depths of computers we use for analysis are high enough to allow for perfect reconstruction between continuous-time signals and digital signals.

We also assume signals collected have been properly sampled according to the Nyquist-Shannon sampling theorem, which states that a discrete-time signal must be sampled at at least twice the highest frequency present in the signal to prevent aliasing of different frequencies. For example, speech signals genearlly have information up to 8kHz, so many speech signals are sampled at 16kHz. Music is more complex in that signals often span up to about 20kHz, so CD quality recordings are often sampled at 44.1kHz or higher. For this thesis, we use recordings sampled at 44.1kHz or lower.

2.3.2 Convolution

The discrete-time convolution operation takes two sequences $x[n]$ and $h[n]$ and outputs a third sequence $y[n] = x[n] * h[n]$:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{15}$$

Convolution is commutative, so $x[n] * h[n] = h[n] * x[n]$ holds true.

A linear, time-invariant (LTI) system is characterized by its impulse response $h[n]$, which allows us to determine samples $y[n]$ when $x[n]$ is subject to $h[n]$. For the purposes of this thesis, our underlying clean signal $x[n]$ might be subject to the conditions of an acoustic environment $h[n]$ and crowd noise $N[n]$:

$$y[n] = h[n] * x[n] + N[n] \tag{16}$$

In this scenario, our system would attempt to recover $h[n] * x[n]$ and possibly even $x[n]$ if the acoustic environment were deemed "noisy enough" due to echo and reverberation.

One of our proposed systems also incorporates convolutional neural networks (CNN) which use convolutions between frames of samples instead of simple linear combinations (discussed later).

2.3.3 Frequency Transforms

In some of our proposed systems, we use a frequency transformed version of the input signal as a preprocessing step to the system input. While no new information is gained from transforming the input, networks often respond

better to determining the value of the magnitude of varying frequencies at a time slice instead of the individual time samples.

The frequency transform we use in this thesis is the discrete-time Fourier transform (DTFT). A sequence of N discrete-time samples is transformed into another sequence of N samples where each index then corresponds to a frequency bin. The DTFT $X[k]$ of a signal $x[n]$ is given by the following:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{17}$$

where the twiddle factor W_N is given by $W_N = e^{-j(2\pi/N)}$. Then the reconstruction of $x[n]$ from $X[k]$ is given by:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn} \tag{18}$$

In this thesis, we also exploit the main duality between the time and frequency domain using the convolution theorem, which states that convolution in time is equivalent to multiplication in frequency and vise versa:

$$\mathscr{F}\{h[n] * x[n]\} = H[k]X[k] \tag{19}$$

$$\mathscr{F}^{-1}\{H[k] * X[k]\} = h[n]x[n] \tag{20}$$

This allows us to effectively treat our network as a non-linear filter that can denoise small time/frequency slices of our noisy signal, which can then be pieced back together using overlap-add resynthesis. We detail this in the next section.

2.3.4 Windowing and Perfect Reconstruction

To window a signal is to multiply a window function $w[n]$ by the frame, i.e. $w[n]x[n]$ over the frame length N. Because we are training a network to denoise small segments of a larger audio signal, we window the signal segments. This accommodates the finite-length requirement of the DTFT and helps to prevent spectral leakage. [7]

Also, to be able to properly reconstruct our signal, we use a window function and corresponding overlapping frame percentage to accomplish perfect reconstruction. The corresponding overlapping frame percentage is set such that the window sums to a constant for all time. For example, a rectangular window $w[n] = 1$ over an interval of length N has an overlap of 0% to sum to a constant 1 for all time. Another popular window is the Hanning window, defined over an interval N by the following:

$$w[n] = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{21}$$

For the Hann window, the perfect reconstruction overlap is a frame length of $N = 50\%$.

2.3.5 Window Size and Frequency v. Time Resolution Tradeoff

We must consider window size as a hyperparameter to our system. In general, shorter windows give rise to better time resolution at the cost of frequency resolution. On the other hand, longer windows give rise to better frequency resolution at the cost of time resolution. To illustrate, consider FIGURE.

2.3.6   Noise and Signal-to-Noise Ratio

Since we are trying to denoise audio signals, we must discuss how we measure noise. One of the most common measures of degradation of signal quality from additive noise is signal-to-noise ratio (SNR), defined as the ratio of signal variance to noise variance. [DSP] For the signal $y[n] = x[n] + N[n]$, where $x[n]$ is the signal of interest and $N[n]$ is the additive noise, the SNR is defined as

$$SNR = \frac{\sigma_x^2}{\sigma_n^2} \tag{22}$$

where σ^2 refers to the variance of the signal in question over some time interval. For the purposes of this thesis, we achieve desired a desired SNR for a simulation by scaling the noise to match the variance to the signal, then scaling the noise or the signal to achieve the desired SNR.

3 Signal Model and Data

3.1 Network Input and Output

To simulate an audio denoising scheme, we define the following inputs and outputs. We take a known clean signal $x[n]$ which we subject to additive noise $N[n]$ using a specified SNR, resulting in the following noisy signal $y[n]$:

$$y[n] = x[n] + N[n] \tag{23}$$

To achieve a particular average SNR per simulation, we take the average signal energy for each minibatch of size B to determine a multiplicative scale factor k on the noise signal $N[n]$. For example, for additive white Gaussian noise (AWGN), we sample from the zero-mean, unit variance normal distribution ("randn" in Python) and determine our scale factor k as σ using the specified SNR in decibels:

$$\sigma_n^2 = \frac{1}{SNR_{lin}} \frac{1}{BN} \sum_{b=0}^{B-1} \sum_{n=0}^{N-1} x_b^2[n] \tag{24}$$

where SNR_{lin} is given by

$$SNR_{lin} = 10^{\frac{SNR_{db}}{10}} \tag{25}$$

In supervised scenarios, we allow the network to train with access to the ground truth $x[n]$. On the other hand, in semi-supervised scenarios, we only allow the network to train with access to a "soft label" indicating if the signal is (1) noise-only or (2) noise and possibly signal. [1] However, in both supervised and semi-supervised scenarios, our neural network input is one of the following:

1. Frames of $y[n]$

2. Frames of $\|Y[k]\|$

3. Magnitude spectroram frames of $Y[k]$

4. Complex spectrogram frames of $Y[k]$

We choose the frame length L, time-domain window $w[n]$, and frame overlap percentage p as hyperparameters. Generally, we use 1024-sample frames at 16 kHz with a Hanning window with 50% overlap unless otherwise specified. In addition, for frequency frames, we use an FFT length the same length as our frame for a total of $L/2$ frequency bins. Note that our choice of frame length and sampling rate allows us to balance time and frequency resolution. With the given frame length and sampling rate, we achieve a frequency resolution of 15.625 Hz/bin by the following:

$$\frac{f_s/2 \text{ Hz}}{N/2 \text{ bins}} = \frac{fs}{N} \tag{26}$$

$$= 15.625 \text{ Hz/bin} \tag{27}$$

Similary, our time resolution is given by

$$\frac{N}{fs} = 64 \text{ msec} \tag{28}$$

Since we want to evaluate the level of denoising in the time domain, we recombine the network outputs with the noisy phase components of the spectrum if necessary to obtain an estimate $\hat{x}[n]$. We then compare $\hat{x}[n]$ to $x[n]$, in general using the mean squared error (MSE). For example, when our network outputs

frames of $\|\hat{X}[k]\|$, we take the inverse Fast Fourier transform (IFFT) using the noisy phase $\angle Y[k]$ and use overlap-add to recombine the frames:

$$x\hat{}[n] = \mathscr{F}^{-1}\{\|\hat{X}[k]\|e^{j\angle Y[k]}\} \qquad (29)$$

3.2   Signal and Noise Choices

Our choice of signals include the following:

1. Sine waves with multiple frequencies and random amplitudes and phases

2. Clean speech signals

3. Studio music recordings

4. Live concert recordings

Similarly, our choice of noise signals include the following:

1. Additive white Gaussian noise (AWGN)

2. Restaurant noise

As mentioned above, we can use the average energy per minibatch to specify a given SNR for an experiment. We take several combinations of clean and noise signals and compare across multiple SNRs.

3.3   Other Network Parameters

Since our networks involve one or more neural network layers, we show some results compared to choices of nonlinearity, number of layers (depth), and number of nodes in each layer (width). Generally, we use an identity at the

network output and either the rectified linear unit (ReLU), a modified ReLU (mReLU), leaky rectify, hyperbolic tangent (tanh), or an exponential linear unit (elu).

4   De-noising Architectures

In the following sections, we detail all considered shallow network architectures. Note that these network architectures can easily be extended to deep networks by adding corresponding encode and decode layers before and after the latent representation, respectively. These networks can be trained using the various inputs detailed in Chapter 3. However, for the purposes of presenting first results, we consider single FFT frames only to compare networks.

4.1   Supervised Autoencoder

We adopt the shallow supervised autoencoder from [2]. Used for supervised denoising, we adopt the relative network size as well as their modified nonlinear activation function. The network structure is a single hidden layer, dense neural network. In other words, we can represent our network output $\hat{X}_i[k]$ for various overlapping frames $i = 1, \cdots, N$ by the following:

$$\hat{X}_i[k] = f_1\big(\mathbf{W}^{(1)}\mathbf{h}_i^{(0)} + \mathbf{b}^{(1)}\big) \tag{30}$$

$$\mathbf{h}_i^{(0)} = f_0\big(\mathbf{W}^{(0)}Y_i[k] + \mathbf{b}^{(0)}\big) \tag{31}$$

This network is trained to estimate the various layer weight matrices $\mathbf{W}^{(l)}$ and layer bias vectors $\mathbf{b}^{(l)}$.

Since we are estimating a magnitude spectrogram for values in the interval $[0, \infty)$, we use a nonlinear activation function whose support is on the same interval. A natural choice is the rectified linear unit (ReLU). However, as detailed in [2], the ReLU is subject to a 0-derivative for negative values. The modified ReLU used in [2], which we denote as mReLU, is given by the following:

$$f(x) = \begin{cases} x & \text{if } x \geq \epsilon \\ \dfrac{-\epsilon}{x - 1 - \epsilon} & \text{if } x < \epsilon \end{cases} \qquad (32)$$

The choice of ϵ used in [2] is 10^{-5}. This modified ReLU allows for nodes to escape zero state since the derivative is always positive. An example plot of the nonlearity is given in 2.
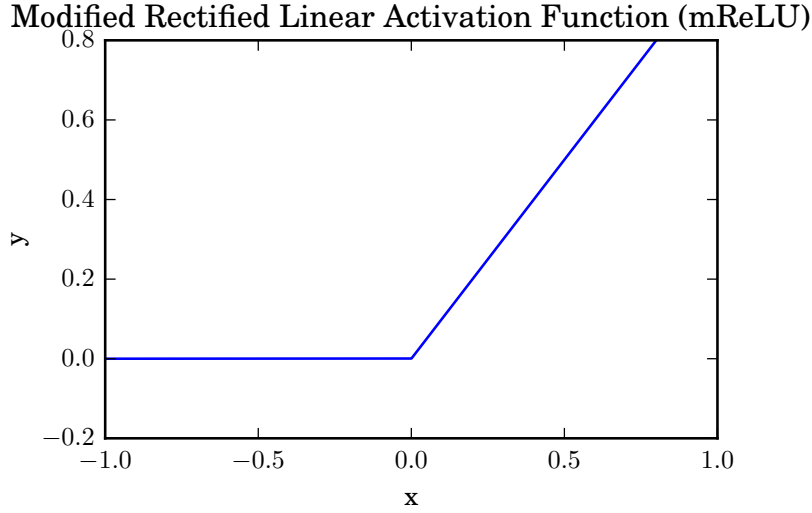

Figure 2: Modified Rectified Linear Unit Activation Function Plot

Since this network is supervised, we allow the training access to the original magnitude spectra $X[k]$. The loss function for training this network is defined as the mean squared error (MSE) between the network output and the clean spectra $X[k]$:

$$l(\mathbf{X}, \widehat{\mathbf{X}}) = \|\mathbf{X} - \widehat{\mathbf{X}}\|^2 \qquad (33)$$

For our simulations, we also apply batch normalization at the input to help train more quickly and efficiently.

4.2 Partitioned Autoencoder

Adopted from [1], the partitioned autoencoder is a variation of a traditional autoencoder in which we do not know the noise corruption process or the underlying clean signals directly. This model more closely models a practical scenario. Since we don't have access to clean data, we rely instead on a "soft" label indicating whether we have a "noise-only" training example or a "noisy" training example which possibly has the desired signal present within it.

Depending on a number of factors, a traditional autoencoder can learn many different latent representations which ultimately learn to encode and decode the underlying clean signal. A partitioned autoencoder seeks to use regularization during training to give explicit meaning to the latent variables in the network. If we can identify noise-only components in our training data, we can potentially train the network to put noise-only information into one part of the latent space. Then, the rest of the latent variables should correspond to signal-only if a sufficient representation of the noise is learned. At inference time, we can then zero out the noise-only latent variables to accomplish denoising.

In [1], they use the following loss function to accomplish effective partitioning:

$$l(\mathbf{Y}, y) = \|\mathbf{Y} - \widehat{\mathbf{X}}\|^2 + \frac{\lambda y}{\overline{\mathbf{C}}}\|\mathbf{C} \odot f(\mathbf{Y})\|^2 \tag{34}$$

where $\widehat{\mathbf{X}} = g(f(\mathbf{Y}))$, the latent variables are given by $f(\mathbf{Y})$, $\mathbf{C}$ is a masking matrix dependent on the minibatch and latent sizes taking on the value 1 for signal latents and 0 for noise or background latents. y corresponds to the aforementioned soft label which has value 0 for a signal-plus-noise example and 1 for a noise-only example. λ is a regularization coefficient which is set to

a higher value than normally used for regularization to enforce zeroing out of signal-based latents for noise-only examples.

In other words, when we train the network, we use minibatches with fixed ordering corresponding to the same proportion of signal-plus-noise examples and noise-only examples such that $\mathbf{C}$ does not change on each training iteration. An example $\mathbf{C}$ is given in Figure 3. For signal-plus-noise examples, the regularization term is 0, and the network seeks to reconstruct the noisy example. However, for noise-only examples, the network tries to put all of the latent energy into the pre-determined noise-only latent variables. In [1], they balance this by choosing 25% of minibatches to be noise-only as well as 25% of latent variables to be noise-only.
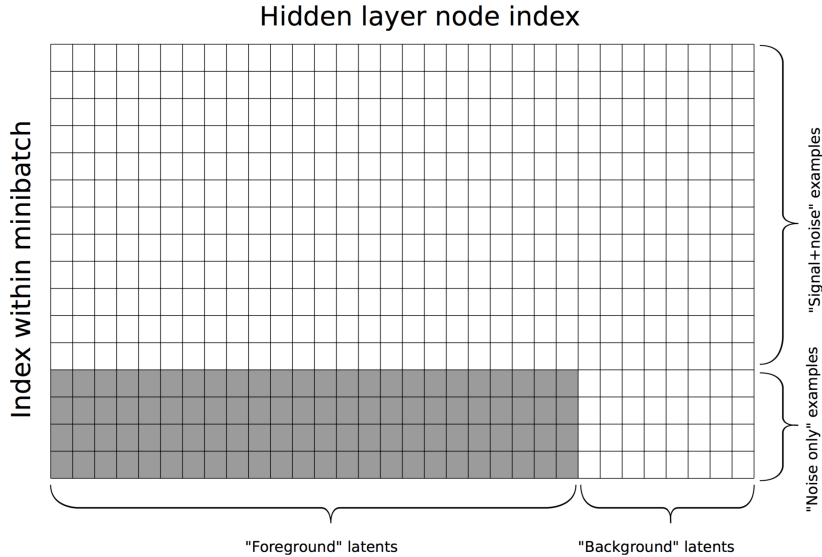

Figure 3: Example Partitioned Masking Matrix. [1] The gray area corresponds to signal (foreground) latents for noise-only examples. We want to penalize the network for any nonzero energy in the signal latents when there are noise-only examples.

Note that it is okay for noise-only examples to be mislabeled as signal-plus-noise, but the opposite would cause the signal to be misrepresented as noise.

Therefore, the soft labeling of examples should be cautious on the side of labeling as noise-only.

In [1], they use spectrogram frames as input. Their partitioned autoencoder is constructed as a shallow two-dimensional convolutional autoencoder with input normalization to zero-mean and unit-variance, maxpooling along the time index, and a ReLU nonlinearity before the latent layer. Their convolutional layer is constructed such that the frequency space is fully connected, and the convolution happens in time. The results of their autoencoder are presented in the frequency domain only, so we seek to adapt the partitioning concept to also recover cleaner time-domain audio.

Therefore, we use the same loss function as defined in Equation 34, but we use single magnitude spectrum frames and compare results on the mean squared error in the time-domain rather than in the frequency domain as their results presented. We also used the modified ReLU as presented from [2].

4.2.1 Phase Reconstruction

By extension, we combine the estimated magnitude spectra at the output with the original noisy phase. We can then recover a time-domain estimate of our desired signal using overlap-add resynthesis.

Other experiments we tried involved trying to explicitly or implicitly estimate the clean phase. One such explicit experiment involved training a parallel partitioned autoencoder with modified nonlinearities that tried to learn a clean phase representation. However, this ended up with a worse MSE and distorted the signal than using the original noisey phase. An implicit phase estimation example involved training the network using two channels as feature maps, where the real part of the frequency spectrum made up one feature map and

23

the imaginary part of the frequency spectrum made up the other. This also resulted in worse reconstructed signals than the case where we estimate the magnitude spectrum and combine with the noisy phase. Sample code is provided in the appendix.

4.3   Curro Autoencoder

We present here the Curro Partitioned Autoencoder, a novel partitioned neural network architecture. Similar to [1], we exploit the latent space structure to put noise and signal energy into different latent variables. A basic overview of the network is detailed in Figure 4.
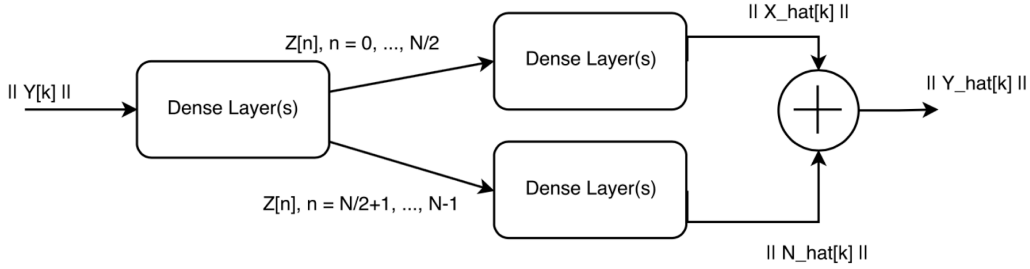
Figure 4: Curro Autoencoder Block Diagram. Partitioning occurs on 50% of the latent space for the signal and the noise. Either can be reconstructed.

In the shallow case, we have an input layer, a fully connected hidden layer, and then a split in the latent space. We split the network such that half of the latent variables correspond to signal and the other half correspond to noise, and then the outputs from both network partitions are summed. For the shallow case, we use one fully connected layer followed by an output layer of the same size. The parallel networks are the same size and share the same parameters $\mathbf{W}$ and $\mathbf{b}$. Unlike in [1], we constrain the problem to 50% of latent variables for signal content and 50% for noise content. While there may be

drawbacks to such a restriction, the benefit here is that we do not have to choose that ratio as a hyperparameter.

More formally,

$$\hat{Y}_i[k] = \hat{X}_i[k] + \hat{N}_i[k] \tag{35}$$

where

$$\hat{X}_i[k] = \mathbf{W}^{(3)} f(\mathbf{W}^{(2)} \mathbf{z}_{i,sig} + \mathbf{b}^{(2)}) + \mathbf{b}^{(3)} \tag{36}$$

$$\hat{N}_i[k] = \mathbf{W}^{(3)} f(\mathbf{W}^{(2)} \mathbf{z}_{i,noi} + \mathbf{b}^{(2)}) + \mathbf{b}^{(3)} \tag{37}$$

and the partitioned latent space $\mathbf{z_i}$ is given by

$$\mathbf{z}_i = f(\mathbf{W}^{(1)} f(\mathbf{W}^{(0)} \mathbf{Y}_i[k] + b^{(0)}) + b^{(1)}) \tag{38}$$

with associated partitions

$$\mathbf{z_i} = \left[\begin{array}{c} \mathbf{z}_{i,sig} \\ \hline \mathbf{z}_{i,noi} \end{array} \right] \tag{39}$$

Note that for a latent space $\mathbf{z}$ with N dimensions, the latent partitions $\mathbf{z}_{i,sig}$ and $\mathbf{z}_{i,noi}$ have dimension $N/2$.

We train the network as in [1] with minibatches consisting of noise-only examples and signal-plus-noise examples. The way we train the network to learn the partitions uses the following loss function:

$$l(\mathbf{Y}, \widehat{\mathbf{X}}) = \begin{cases} \|\mathbf{Y} - \widehat{\mathbf{N}}\|^2 & \text{if } \mathbf{Y} \text{ is noise-only} \\ \|\mathbf{Y} - \widehat{\mathbf{Y}}\|^2 & \text{if } \mathbf{Y} \text{ is signal-plus-noise} \end{cases} \tag{40}$$

We introduce again a soft label y indicating if the example is noise-only or signal-plus-noise. We can then rewrite our loss function as

$$l(\mathbf{Y}, \widehat{\mathbf{X}}) = \text{MSE}(\mathbf{Y}, y\widehat{\mathbf{X}} + \widehat{\mathbf{Y}}) \tag{41}$$

$$= \|\mathbf{Y} - y\widehat{\mathbf{X}} - \widehat{\mathbf{Y}}\|^2 \tag{42}$$

In this case, the soft label y takes on the opposite values as in [1], i.e. $y = 0$ for noise-only examples and $y = 1$ for signal-plus-noise examples.

This network also has the benefit of being able to reconstruct both the noise and the signal independently. At inference time, the desired signal can be obtained by only reconstructing the top half of the network. Also, in the case of introduced distortion, a proportion of the signal half and noise half can be combined at different ratios, i.e. $\widehat{\mathbf{X}} + \alpha\widehat{\mathbf{N}}$. Depending on circumstances, this can be introduced as a learned parameter or can be tuned manually or through a grid search.

5 Results

We present results here for mainly shallow network architectures. At the output layer of each network, an identity nonlinearity is used. At any other layer, the modified ReLU (mReLU) is used. Unless otherwise noted, batch normalization is applied at the input layer. Each network is compared first to itself at varying noise levels (-6 dB, -3 dB, 0 dB, 3 dB, 6 dB SNR) in terms of convergence as well as the mean squared error (MSE) for inferences.

Training minibatches consist of 128 examples, each with 1024-sample FFT frames of $\|Y[k]\|$ at a sampling rate 16 kHz. Time windows are windowed using the Hanning window, and we use 50% overlap for perfect reconstruction at inference time. The examples used are a sum of sine waves at four fixed frequencies with uniform random amplitude and phase. The frequencies are chosen to form an A4 major chord (1-3-5-8) at slightly de-tuned frequencies so as not to allow the network to learn any pattern from the immediate harmonic structure.

$$f = [441, 549, 660, 881] \qquad \text{Hz} \tag{43}$$

$$x[n] = \sum_{i=0}^{3} A_i \sin 2\pi f_i / f_s n + \phi_i, \qquad n = 0 \dots, 1023 \tag{44}$$

$$A_i \sim U(0.25, 0.75) \tag{45}$$

$$\phi \sim U(0, 2\pi) \tag{46}$$

Applied noise is additive-white Gaussian noise (AWGN), with the variance σ^2 selected to achieve the desired average SNR for each minibatch as in Equation 24.

$$N[n] \sim N(0, \sigma^2), \qquad n = 0 \dots, 1023 \tag{47}$$

$$y[n] = x[n] + N[n] \tag{48}$$

In semi-supervised cases where we use the soft label y for noise-only versus signal-plus-noise examples, we use 25% noise-only examples per minibatch. For inference calculations, we construct a minibatch with consecutive overlapping, windowed frames.

Simulations are written in Python 2.7 using Lasasgne [8], a "lightweight library to build and train neural networks in Theano." Theano is a "Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently." [9] Theano boasts parallel GPU support, numpy support (a mathematical Python library), numerical stability, and symbolic differentiation, among other features. These libraries and frameworks allow for ease of developing deep, novel architectures and save time in doing things like calculating gradients, weight updates and back-propagation. Sample simulation code is shown in the Appendix.

Weight updates are calculated using Adam updates [10]. 2000 iterations (minibatches) are used for each simulation. Unless otherwise noted, each hidden layer uses 2000 hidden nodes.

Loss-based plots show the loss function convergence during training iterations, where the l. Mean squared error plots show the MSE every 50 training examples for an inference example that does not change.

5.1 Supervised Autoencoder

The following results show a single-layer autoencoder with and without batch normalization at the input layer.

5.1.1 Batch Normalized Input

In Figure 5, we can see that the loss function appears to converge at or before 2000 iterations. As expected, as SNR increases, the loss objective converges to a lower value. Since this network is trained using only the squared error loss, this should be expected. Note that as the SNR increases, the difference in the converged value gets smaller. Also interesting is the fact that lower SNR plots converge more quickly but to higher values. This suggests that the network does not respond well to too much noise.
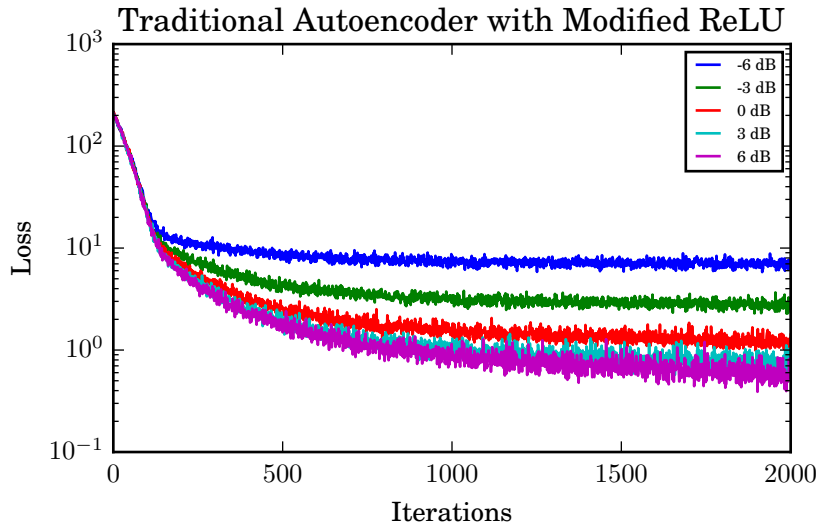

Figure 5: Loss at various SNRs for Supervised Single-Layer Autoencoder with Batch Normalization at the Input

In Figure 6, we can see that the MSE generally goes down as SNR goes up. This should be expected, though perhaps there may be an error in the simulation since the lines blur a bit between -3 dB and 6 dB.

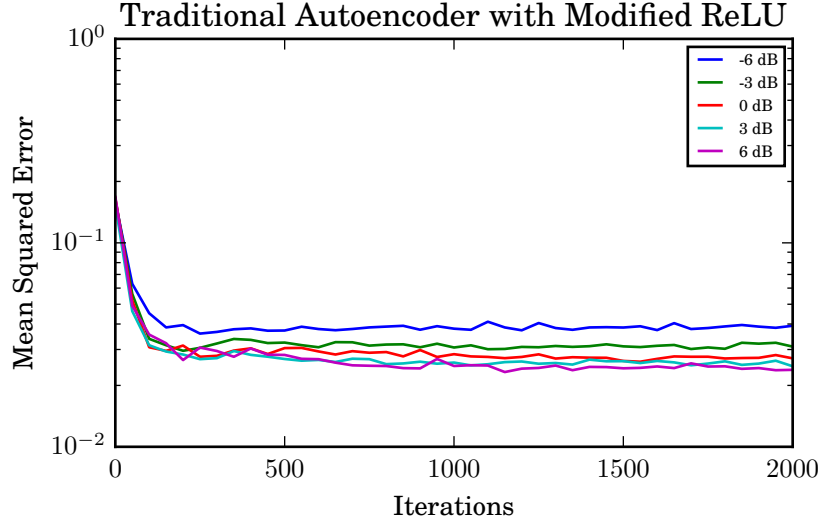Also, the saved audio from which the MSE's were calculated have some distortion introduced from the network.

Figure 6: MSE at various SNRs for Supervised Single-Layer Autoencoder with Batch Normalization at the Input

5.1.2 Non-Batch Normalized Input

As expected, in Figure 7, the loss metric converges about the same as for the batch normalized case. As expected, the convergence time in terms of number of iterations is slightly higher. One interesting section is how the 6 dB curve converges. It appears to have a strong section of downward concavity. It is possible that this occurred randomly, as the random number generator in Numpy was set to a random seed. It is also possible that because of an absence of batch normalization, some neurons saturated and did not change substantially for some time.

Somewhat unexpectedly, the MSE in Figure 8 converges to a lower value than that of Figure 6. This suggests that there may be an error in the simulation, likely in using the stored statistics for batch normalization as opposed to using an on-the-fly calculation of the minibatch statistics at inference time. However,
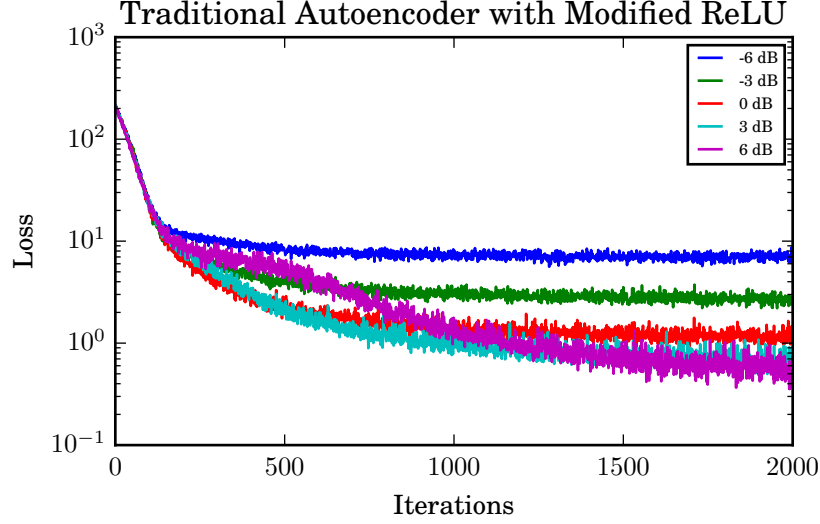
Figure 7: Loss at various SNRs for Supervised Single-Layer Autoencoder without Batch Normalization at the Input

we still achieve convergence here which is expected. Past 0 dB, the MSE seems to converge to a similar value, suggesting that the network has diminishing returns for higher SNR.
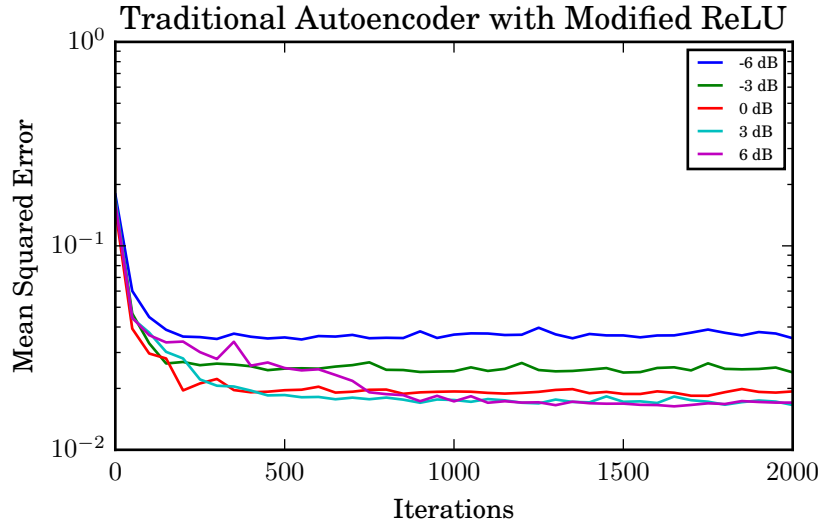

Figure 8: MSE at various SNRs for Supervised Single-Layer Autoencoder without Batch Normalization at the Input
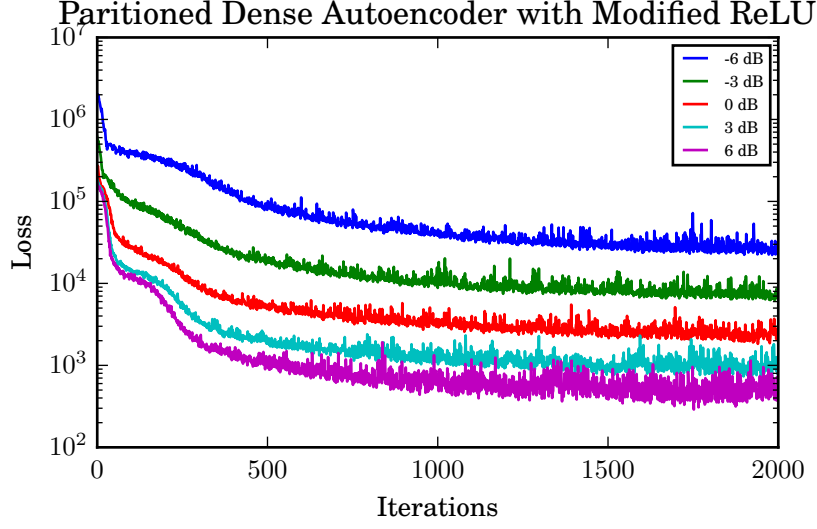
Figure 9: Loss at various SNRs for Single-Layer Partitioned Autoencoder [1]
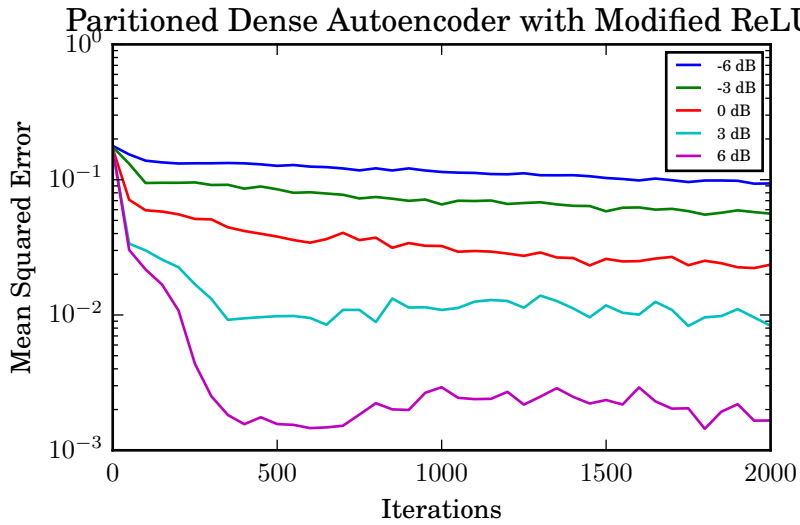

Figure 10: MSE at various SNRs for Single-Layer Partitioned Autoencoder [1]

5.2 Partitioned Autoencoder

5.3 Partitioned Curro Autoencoder

5.4 Comparison of Loss Convergence

5.5 Comparison of Mean Squared Error Convergence

Paritioned Dense Curro Autoencoder with Modified ReLU
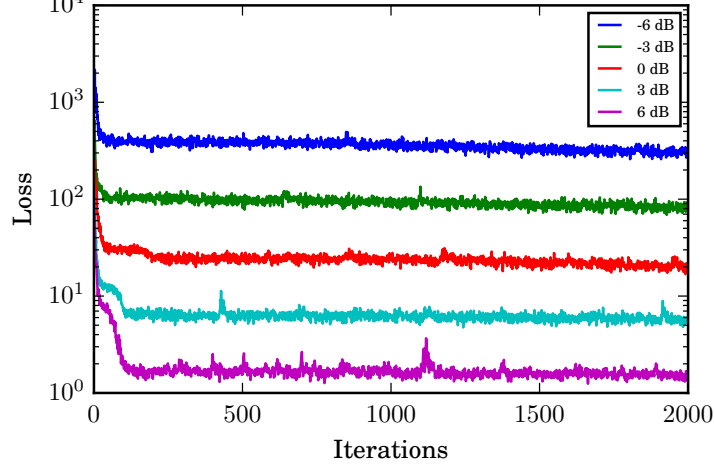
Figure 11: Loss at various SNRs for Single-Layer Curro Autoencoder

Paritioned Dense Curro Autoencoder with Modified ReLU
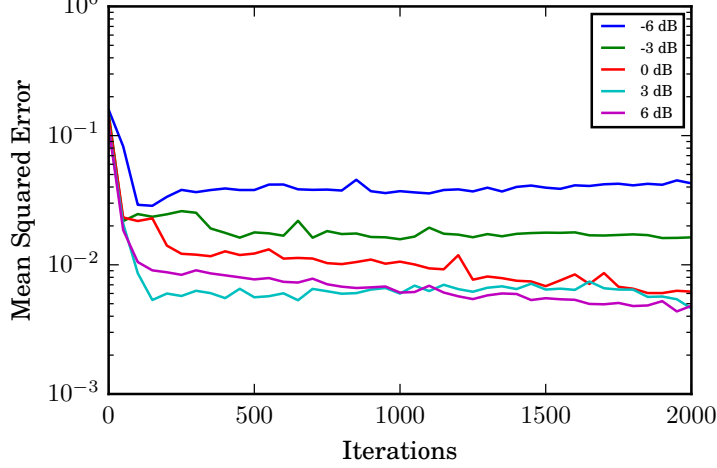
Figure 12: MSE at various SNRs for Single-Layer Curro Autoencoder

6   Conclusions and Future Work

6.1   Conclusions

While more work is needed, deep partitioned neural network architectures using time and frequency data seem promising in long-term solutions for denoising speech and music signals.
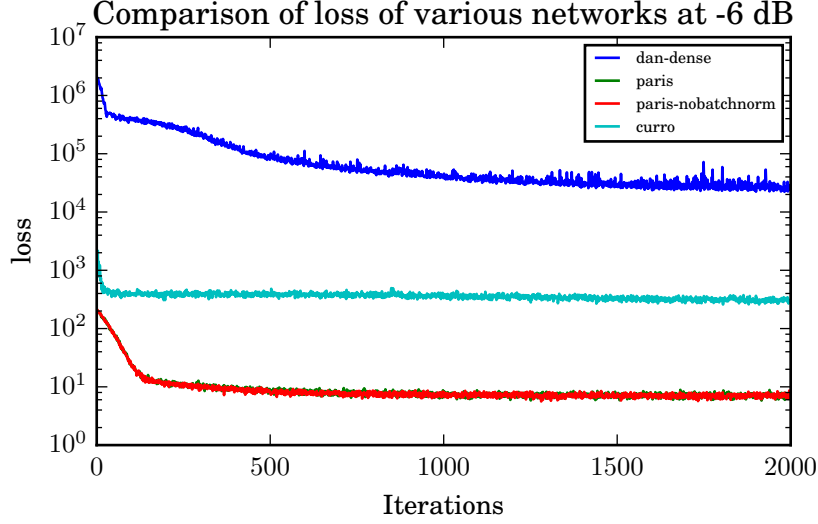
Figure 13: Loss Comparison of Various Networks at -6 dB
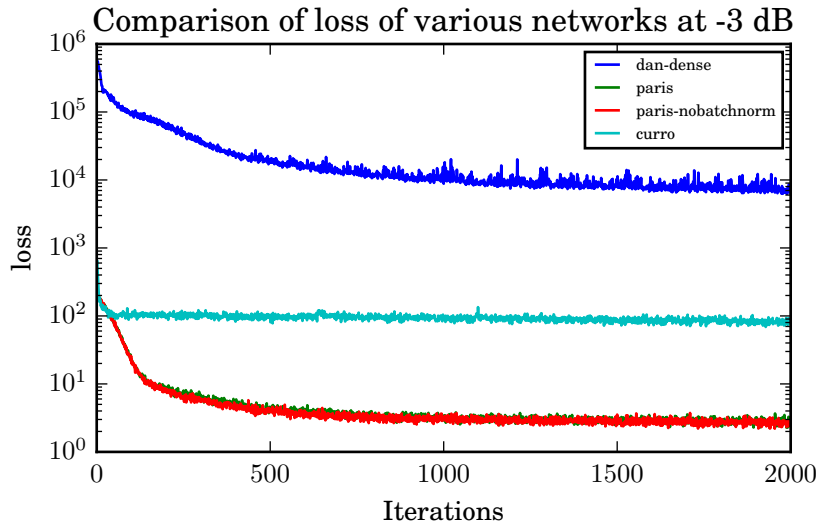
Figure 14: Loss Comparison of Various Networks at -3 dB

6.2 Future Work

6.2.1 Models

Make network deeper. Consider gradual partitioning instead of hard.
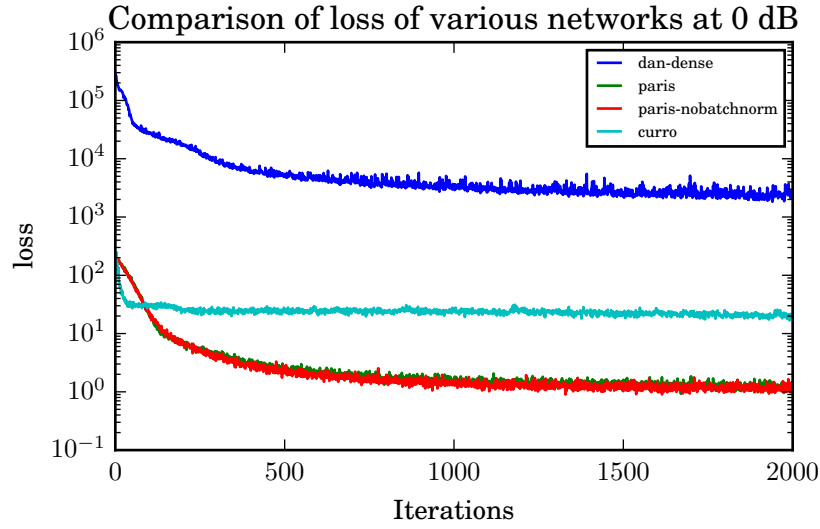
Figure 15: Loss Comparison of Various Networks at 0 dB
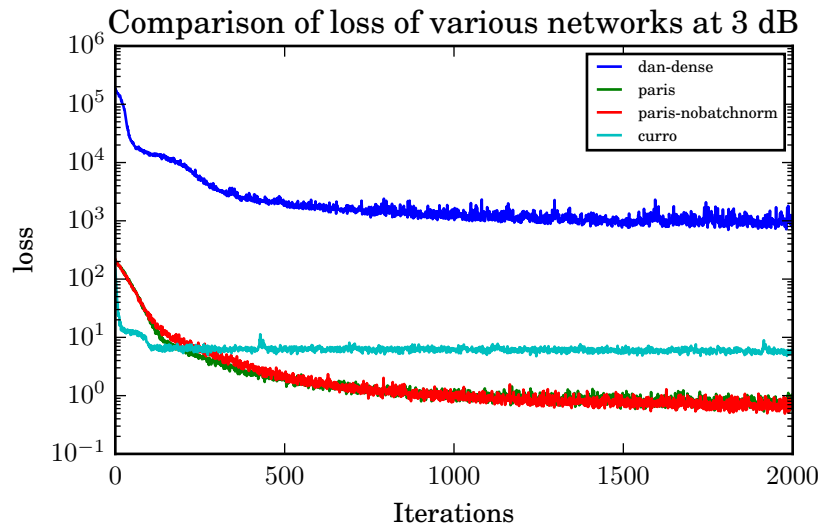
Figure 16: Loss Comparison of Various Networks at 3 dB

6.2.2   Data

Get more data. Consider different noise levels and types of signals.
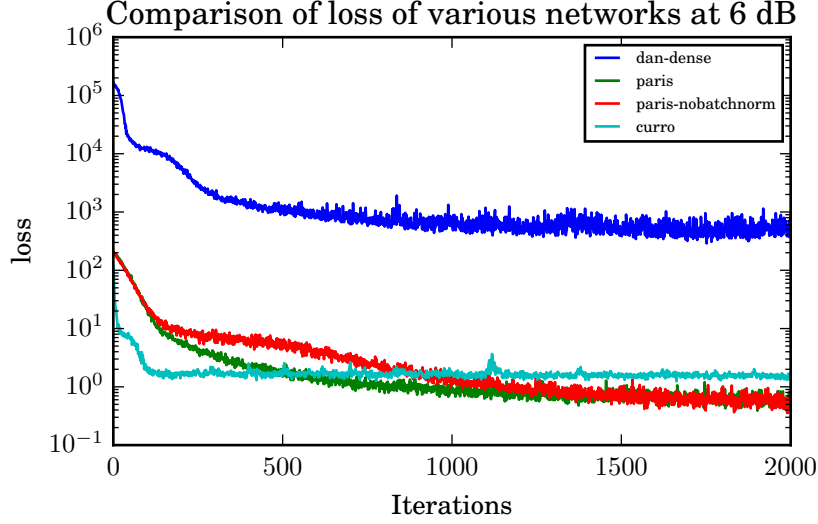
Figure 17: Loss Comparison of Various Networks at 6 dB
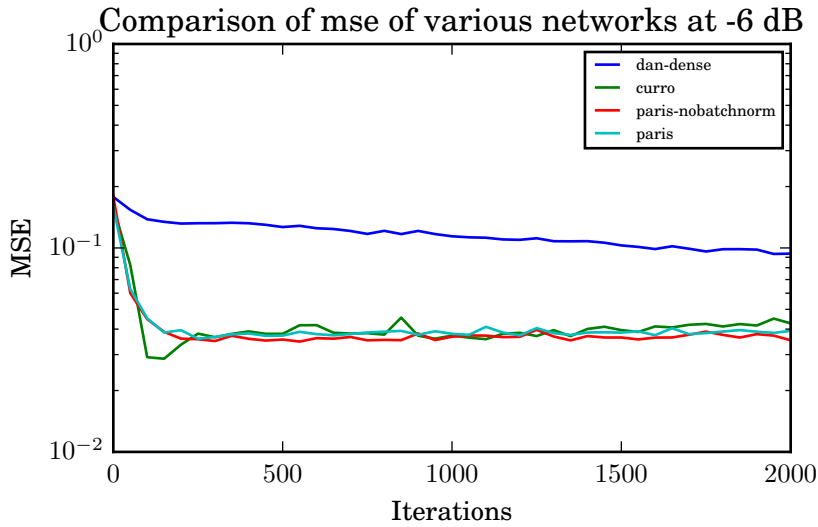
Figure 18: MSE Comparison of Networks at -6 dB

References

[1] D. Stowell and R. E. Turner, "Denoising without access to clean data using a partitioned autoencoder," *CoRR*, vol. abs/1509.05982, 2015. [Online]. Available: http://arxiv.org/abs/1509.05982

[2] D. Liu, P. Smaragdis, and M. Kim, "Experiments on deep learning for speech denoising." in *INTERSPEECH*, 2014, pp. 2685–2689.

[3] C. M. Bishop, *Pattern recognition and machine learning.* springer,
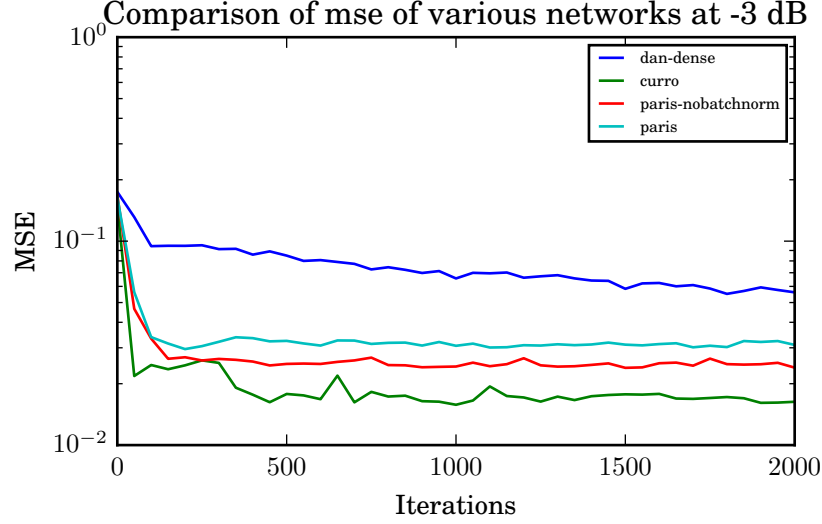
Figure 19: MSE Comparison of Networks at -3 dB
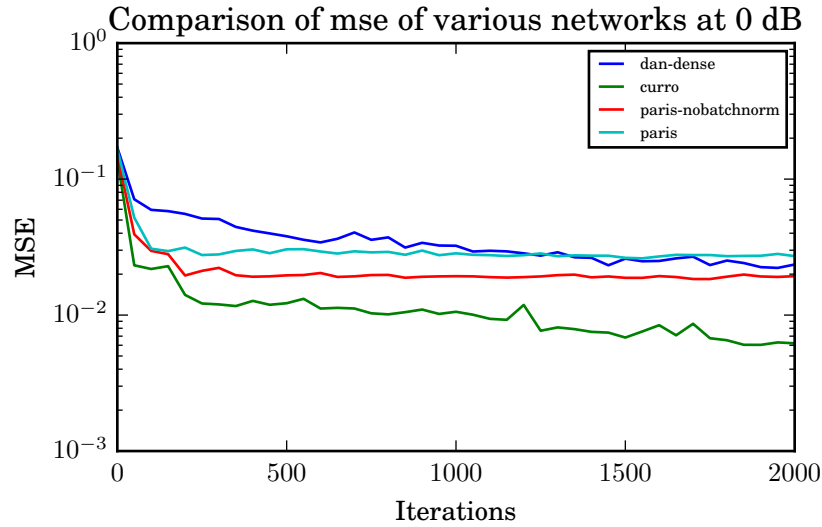
Figure 20: MSE Comparison of Networks at 0 dB

2006.

[4] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks." in *Aistats*, vol. 15, no. 106, 2011, p. 275.

[5] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural Networks*, vol. 16, no. 10, pp. 1429–1451, 2003.

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint*
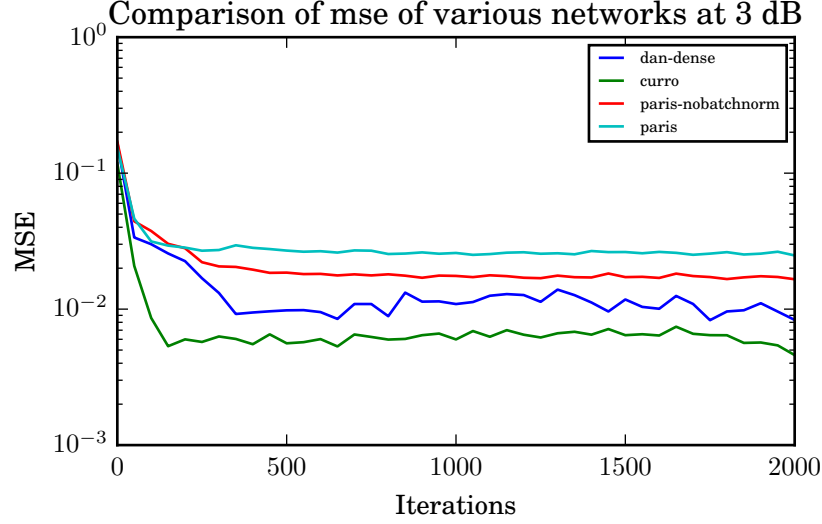
Figure 21: MSE Comparison of Networks at 3 dB
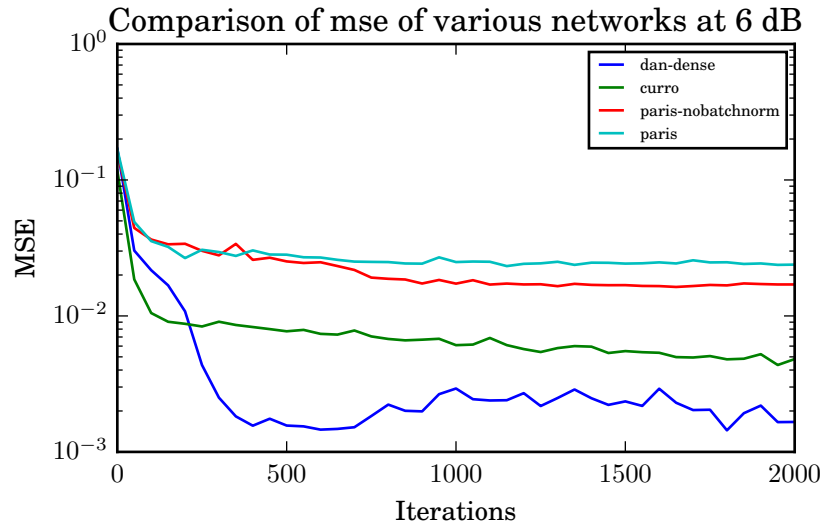
Figure 22: MSE Comparison of Networks at 6 dB

arXiv:1502.03167, 2015.

[7] V. O. Alan, W. S. Ronald, and R. John, "Discrete-time signal processing," *New Jersey, Printice Hall Inc*, 1989.

[8] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, D. Maturana, M. Thoma, E. Battenberg, J. Kelly, J. D. Fauw, M. Heilman, diogo149, B. McFee, H. Weideman, takacsg84, peterderivaz, Jon, instagibbs, D. K. Rasul, CongLiu, Britefury, and

J. Degrave, "Lasagne: First release." Aug. 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.27878

[9] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[11] P. Baldi and Z. Lu, "Complex-valued autoencoders," *Neural Networks*, vol. 33, pp. 136–147, 2012.

[12] Y. Xu, J. Du, L.-R. Dai, and C.-H. Lee, "An experimental study on speech enhancement based on deep neural networks," *IEEE Signal Processing Letters*, vol. 21, no. 1, pp. 65–68, 2014.

[13] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[14] T. Ishii, H. Komiyama, T. Shinozaki, Y. Horiuchi, and S. Kuroiwa, "Reverberant speech recognition based on denoising autoencoder." in *INTERSPEECH*, 2013, pp. 3512–3516.

[15] B. Gold, N. Morgan, and D. Ellis, "Speech and audio signal processing: processing and perception of speech and music," 2011.

[16] M. Kayser and V. Zhong, "Denoising convolutional autoencoders for noisy speech recognition."

[17] U. Zölzer, *Digital audio signal processing.* John Wiley & Sons, 2008.

[18] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[19] A. Rad and T. Virtanen, "Phase spectrum prediction of audio signals," in *International Symposium on Communications Control and Signal Processing (ISCCSP)*, 2012, pp. 1–5.

[20] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, W. W. Cohen, A. McCallum, and S. T. Roweis, Eds. ACM, 2008, pp. 1096–1103.

[21] W. W. Cohen, A. McCallum, and S. T. Roweis, Eds., *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*. ACM, 2008.

[22] (2010) Denoising autoencoders (da). [Online]. Available: http://deeplearning.net/dA.html

[23] S. Sonoda and N. Murata, "Decoding stacked denoising autoencoders," *CoRR*, vol. abs/1605.02832, 2016. [Online]. Available: http://arxiv.org/abs/1605.02832

A Simulation Code

The following code samples show how network architectures were constructed, how GPU functions were compiled, how networks were trained, and how simulation results were collected and plotted.

```python
from __future__ import division
# different networks (autoencoder, conv autoencoder, recurrent)
# different signals (sine, recording)
# different noises (awgn, crowd)
# different domains (time, freq)
from numpy import complex64
import scipy
import lasagne
import theano
import theano.tensor as T
import numpy as np
from scikits.audiolab import wavwrite
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

SIMULATION_SNR = 6
FILE_SNR = '{} dB'.format(SIMULATION_SNR)
FILENAME_LOSS = 'plotfinal/curro-loss.csv'
FILENAME_MSE = 'plotfinal/curro-mse.csv'
LOSSFILE = open(FILENAME_LOSS, 'a')
MSEFILE = open(FILENAME_MSE, 'a')
LINEFMT = FILE_SNR + ',{}\n'
```

```python
23  # for dan net, we look at square loss & reg loss
24  LINEFMTLOSS = FILE_SNR + ',{},{},{}\n'

25

26

27  dtype = theano.config.floatX
28  batchsize = 128
29  srate = 16000
30  pct = 0.5   # overlap
31  fftlen = 1024
32  framelen = fftlen

33

34  # dan-specific
35  shape = (batchsize, framelen)
36  latentsize = 2000
37  background_latents_factor = 0.25
38  minibatch_noise_only_factor = 0.5   # also for curro net
39  n_noise_only_examples = int(minibatch_noise_only_factor * batchsize)
40  n_background_latents = int(background_latents_factor * latentsize)
41  lambduh = 0.75

42

43  batch_norm = lasagne.layers.batch_norm

44

45

46  def mod_relu(x):
47      eps = 1e-5
48      return T.switch(x > eps, x, -eps/(x-1-eps))

49
```

```python
50
51  def normalize(x):
52      return x / max(abs(x))
53
54
55  def snr_after(x, x_hat):
56      return np.var(x)/np.var(x-x_hat)
57
58
59  class ZeroOutBackgroundLatentsLayer(lasagne.layers.Layer):
60
61      def __init__(self, incoming, **kwargs):
62          super(ZeroOutBackgroundLatentsLayer, self).__init__(incoming)
63          mask = np.ones((batchsize, latentsize))
64          mask[:, 0:n_background_latents] = 0
65          self.mask = theano.shared(mask, borrow=True)
66
67      def get_output_for(self, input_data, reconstruct=False, **kwargs):
68          if reconstruct:
69              return self.mask * input_data
70          return input_data
71
72
73  def dan_net():
74      # net
75      x = T.matrix('X')  # input
76      y = T.matrix('Y')  # soft label
```

```python
77    network = batch_norm(lasagne.layers.InputLayer(shape, x))
78    # network = lasagne.layers.InputLayer(shape, x)
79    print network.output_shape
80    network = lasagne.layers.DenseLayer(
81        network, latentsize, nonlinearity=mod_relu)
82    print network.output_shape
83    latents = network
84    network = ZeroOutBackgroundLatentsLayer(
85        latents, background_latents_factor=background_latents_factor)
86    network = lasagne.layers.DenseLayer(
87        network, shape[1], nonlinearity=lasagne.nonlinearities.rectify)
88    print network.output_shape
89
90    # loss
91    C = np.zeros((batchsize, latentsize))
92    C[0:n_noise_only_examples, n_background_latents + 1:] = 1
93    C_mat = theano.shared(np.asarray(C, dtype=dtype), borrow=True)
94    mean_C = theano.shared(C.mean(), borrow=True)
95    prediction = lasagne.layers.get_output(network)
96    mse_term = lasagne.objectives.squared_error(
97        prediction, x).sum(axis=[1], keepdims=True)
98    scf = lambduh/mean_C
99    regularization_term = scf * y * \
100       ((C_mat * lasagne.layers.get_output(latents))
101        ** 2).sum(axis=[1], keepdims=True)
102   loss = mse_term + regularization_term
103   loss = loss.mean()
```

44

```python
104
105        # training compilation
106        params = lasagne.layers.get_all_params(network, trainable=True)
107        updates = lasagne.updates.adam(loss, params)
108        train_fn = theano.function([x, y], loss, updates=updates)
109
110        # inference compilation
111        predict_fn = theano.function(
112            [x], lasagne.layers.get_output(network, deterministic=True, reconstruct=Tr
113
114        #
115        # other objectives
116        #
117        square_term = theano.function([x], mse_term.mean())
118        regularization_term = theano.function([x, y], regularization_term.mean())
119
120        def do_stuff(network, latents, predict_fn):
121            pass
122
123        return network, latents, loss, square_term, regularization_term, train_fn, pre
124
125
126    def dan_main(params):
127        network, latents, loss, square_loss, reg_loss, train_fn, predict_fn, do_stuff
128        lmse = []
129        lsq = []
130        lreg = []
```

45

```python
131        # inference example for simulations
132    clean, noisy, n, labels = gen_freq_data(
133        sample=True, gen_data_fn=gen_batch_half_noisy_half_noise)
134    for i in xrange(params.niter+1):
135        _clean, _noisy, _n, _labels = gen_freq_data(
136            sample=False, gen_data_fn=gen_batch_half_noisy_half_noise)
137        # swap 0 and 1 since for dan net, 0 is signal and 1 is background
138        _labels = np.expand_dims(np.abs(_labels-1).astype(dtype)[:, 1], axis=1)
139        # labels = np.abs(labels-1).astype(dtype)
140
141        loss = train_fn(_noisy[0], _labels)
142        lmse.append(loss)
143
144        loss_lsq = square_loss(_noisy[0])
145        lsq.append(loss_lsq)
146
147        loss_reg = reg_loss(_noisy[0], _labels)
148        lreg.append(loss_reg)
149        print '%d\t%.3E\t%.3E\t%.3E' % (i, loss, loss_lsq, loss_reg)
150
151        LOSSFILE.write(LINEFMTLOSS.format(loss, loss_lsq, loss_reg))
152
153        if i in range(0, params.niter+50, 50):
154            # validate mse
155            cleaned_up = predict_fn(noisy[0])
156            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
157            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
```

```
158            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))

159            baseline_mse = mean_squared_error(clean_time, noisy_time)

160            print 'baseline mse:', baseline_mse

161            mse = mean_squared_error(cleaned_up_time, clean_time)

162            print 'mse:', mse

163            MSEFILE.write(LINEFMT.format(mse))

164

165        cleaned_up = predict_fn(noisy[0])

166        print 'freq mse:', mean_squared_error(cleaned_up, clean[0])

167        cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))

168        cleaned_up_clean_phase = normalize(ISTFT(cleaned_up, clean[1], fftlen))

169        clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))

170        noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))

171        print 'time mse noisy phase:', mean_squared_error(cleaned_up_time, clean_time)

172        print 'time mse clean phase:', mean_squared_error(cleaned_up_clean_phase, clea

173        print 'baseline time mse noisy to clean:', mean_squared_error(noisy_time, clea

174        wavwrite(clean_time, 'dan/x.wav', fs=srate, enc='pcm16')

175        wavwrite(noisy_time, 'dan/y.wav', fs=srate, enc='pcm16')

176        wavwrite(cleaned_up_time, 'dan/xhat.wav', fs=srate, enc='pcm16')

177        wavwrite(cleaned_up_clean_phase,

178                'dan/xhat_cleanphase.wav', fs=srate, enc='pcm16')

179

180        plt.figure()

181        plt.semilogy(lmse)

182        plt.semilogy(lsq)

183        plt.semilogy(lreg)

184        plt.legend(['overall loss', 'squared error loss', 'regularization loss'])
```

```python
185         plt.savefig('dan/losses.svg', format='svg')
186
187
188     def paris_net(params):
189         shape = (batchsize, fftlen)
190         x = T.matrix('x')   # dirty
191         s = T.matrix('s')   # clean
192         #in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))
193         in_layer = lasagne.layers.InputLayer(shape, x)
194         h1 = batch_norm(
195             lasagne.layers.DenseLayer(in_layer, 2000, nonlinearity=mod_relu))
196         h1 = lasagne.layers.DenseLayer(
197             h1, fftlen, nonlinearity=lasagne.nonlinearities.identity)
198
199         # loss function
200         prediction = lasagne.layers.get_output(h1)
201         loss = lasagne.objectives.squared_error(prediction, s)
202         return h1, x, s, loss.mean(), None, prediction
203
204
205     def curro_net(params):
206         # input
207         shape = (batchsize, framelen)
208         x = T.matrix('x')   # dirty input
209         label = T.matrix('label')   # noise OR signal/noise
210
211         nonlin = mod_relu
```

```python
# network
# in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))  # batch norm
# or no?
in_layer = lasagne.layers.InputLayer(shape, x)  # batch norm or no?
layersizes = 1024*2
h1 = lasagne.layers.DenseLayer(in_layer, layersizes, nonlinearity=nonlin)
h2 = lasagne.layers.DenseLayer(h1, layersizes, nonlinearity=nonlin)
h3 = lasagne.layers.DenseLayer(h2, layersizes, nonlinearity=nonlin)
f = h3  # at this point, first half is signal, second half is noise

# signal split
f_sig = lasagne.layers.SliceLayer(
    f, indices=slice(0, int(layersizes/2)), axis=-1)
print 'sig split size: ', lasagne.layers.get_output_shape(f_sig)
sig_d3 = lasagne.layers.DenseLayer(f_sig, framelen, nonlinearity=nonlin)
# save parameters for noise split
d3_W = sig_d3.W
d3_b = sig_d3.b
sig_d2 = lasagne.layers.DenseLayer(sig_d3, framelen, nonlinearity=nonlin)
d2_W = sig_d2.W
d2_b = sig_d2.b
g_sig = lasagne.layers.DenseLayer(
    sig_d2, framelen, nonlinearity=lasagne.nonlinearities.identity)
gs_W = g_sig.W
gs_b = g_sig.b
```

```python
        f_noi = lasagne.layers.SliceLayer(
            f, indices=slice(int(layersizes/2), layersizes), axis=-1)
        print 'noisy split size: ', lasagne.layers.get_output_shape(f_noi)
        noi_d3 = lasagne.layers.DenseLayer(
            f_noi, framelen, W=d3_W, b=d3_b, nonlinearity=nonlin)
        noi_d2 = lasagne.layers.DenseLayer(
            noi_d3, framelen, W=d2_W, b=d2_b, nonlinearity=nonlin)
        g_noi = lasagne.layers.DenseLayer(
            noi_d2, framelen, W=gs_W, b=gs_b, nonlinearity=lasagne.nonlinearities.iden

        out_layer = lasagne.layers.ElemwiseSumLayer([g_sig, g_noi])

        prediction_sig = lasagne.layers.get_output(g_sig)
        prediction_noi = lasagne.layers.get_output(g_noi)
        # label is 1 for signal, 0 for noise
        prediction = label * prediction_sig + prediction_noi
        loss = lasagne.objectives.squared_error(prediction, x)
        loss_sig = lasagne.objectives.squared_error(prediction_sig, x)
        loss_noi = lasagne.objectives.squared_error(prediction_noi, x)

        return out_layer, g_sig, x, label, loss.mean(), g_noi, prediction, loss_sig, l


    def autoencoder(params):
        # network
        shape = (batchsize, framelen)
        x = T.matrix('x')  # dirty
```

```python
266        s = T.matrix('s')  # clean
267        in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))
268        h1 = batch_norm(lasagne.layers.DenseLayer(
269            in_layer, 400, nonlinearity=lasagne.nonlinearities.leaky_rectify))
270        h2 = batch_norm(lasagne.layers.DenseLayer(
271            h1, 330, nonlinearity=lasagne.nonlinearities.leaky_rectify))
272        h3 = batch_norm(lasagne.layers.DenseLayer(
273            h2, 300, nonlinearity=lasagne.nonlinearities.leaky_rectify))
274        h4 = batch_norm(lasagne.layers.DenseLayer(
275            h3, 270, nonlinearity=lasagne.nonlinearities.leaky_rectify))
276        bottle = h4
277        d4 = batch_norm(lasagne.layers.DenseLayer(
278            h4, 300, nonlinearity=lasagne.nonlinearities.leaky_rectify))
279        d3 = batch_norm(lasagne.layers.DenseLayer(
280            d4, 330, nonlinearity=lasagne.nonlinearities.leaky_rectify))
281        d2 = batch_norm(lasagne.layers.DenseLayer(
282            d3, 400, nonlinearity=lasagne.nonlinearities.leaky_rectify))
283        x_hat = batch_norm(lasagne.layers.DenseLayer(
284            d2, framelen, nonlinearity=lasagne.nonlinearities.identity))
285
286        # loss function
287        prediction = lasagne.layers.get_output(x_hat)
288        loss = lasagne.objectives.squared_error(prediction, s)
289        reg = 2 * (1e-5 * lasagne.regularization.regularize_network_params(x_hat, lasa
290                   1e-6 * lasagne.regularization.regularize_network_params(x_hat, lasa
291        loss = loss + reg
292        return x_hat, x, s, loss.mean(), reg.mean(), prediction
```

51

```
293

294

295   def train(autoencoder, x, s, loss):

296       params = lasagne.layers.get_all_params(autoencoder, trainable=True)

297       updates = lasagne.updates.adam(loss, params)

298       train_fn = theano.function([x, s], loss, updates=updates)

299       return train_fn

300

301

302   def gen_data(sample=False):

303       def _sin_f(a, f, srate, n, phase):

304           return a * np.sin(2*np.pi*f/srate*n+phase)

305

306       def _noise_var(clean, snr_db):

307           # we use one noise variance per minibatch

308           avg_energy = np.sum(clean*clean)/clean.size

309           snr_lin = 10**(snr_db/10)

310           noise_var = avg_energy / snr_lin

311           print '\tnoise variance for minibatch: ', noise_var

312           return noise_var

313

314       # f = 440

315       if sample:

316           n = np.linspace(0, batchsize * framelen - 1, batchsize * framelen)

317           phase1 = np.random.uniform(0.0, 2*np.pi)

318           phase2 = np.random.uniform(0.0, 2*np.pi)

319           phase3 = np.random.uniform(0.0, 2*np.pi)
```

```python
        phase4 = np.random.uniform(0.0, 2*np.pi)
        amp1 = np.random.uniform(0.25, 0.75)
        amp2 = np.random.uniform(0.25, 0.75)
        amp3 = np.random.uniform(0.25, 0.75)
        amp4 = np.random.uniform(0.25, 0.75)
    else:
        n = np.tile(np.linspace(0, framelen-1, framelen), (batchsize, 1))
        phase1 = np.tile(
            np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
        phase2 = np.tile(
            np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
        phase3 = np.tile(
            np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
        phase4 = np.tile(
            np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
        amp1 = np.tile(
            np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
        amp2 = np.tile(
            np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
        amp3 = np.tile(
            np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
        amp4 = np.tile(
            np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
    # clean = amp * np.sin(2 * np.pi * f / srate * n + phase)
    clean = _sin_f(amp1, 441, srate, n, phase1) + \
        _sin_f(amp2, 549, srate, n, phase2) + \
        _sin_f(amp3, 660, srate, n, phase3) + \
```

```python
347             _sin_f(amp4, 881, srate, n, phase4)

348

349         # corrupt with gaussian noise
350         var = _noise_var(clean, SIMULATION_SNR)
351         noise = np.random.normal(0, var, clean.shape)
352         noisy = clean + noise

353

354         if sample:
355             noisy = np.array(
356                 [noisy[i:i+framelen] for i in xrange(0, len(noisy), int(pct*framelen))
357             clean = np.array(
358                 [clean[i:i+framelen] for i in xrange(0, len(clean), int(pct*framelen))
359             #noisy = noisy.reshape(batchsize, framelen)
360             #clean = clean.reshape(batchsize, framelen)

361

362         return clean.astype(dtype), noisy.astype(dtype), n, None

363

364

365     def gen_batch_half_noisy_half_noise(sample=False):
366         def _sin_f(a, f, srate, n, phase):
367             return a * np.sin(2*np.pi*f/srate*n+phase)

368

369         nop = minibatch_noise_only_factor  # noise only percentage of minibatch
370         f = 440
371         if sample:
372             n = np.linspace(0, batchsize * framelen - 1, batchsize * framelen)
373             np.random.seed(3)  # to get consistent samples
```

54

```python
374         phase1 = np.random.uniform(0.0, 2*np.pi)
375         phase2 = np.random.uniform(0.0, 2*np.pi)
376         phase3 = np.random.uniform(0.0, 2*np.pi)
377         phase4 = np.random.uniform(0.0, 2*np.pi)
378         amp1 = np.random.uniform(0.25, 0.75)
379         amp2 = np.random.uniform(0.25, 0.75)
380         amp3 = np.random.uniform(0.25, 0.75)
381         amp4 = np.random.uniform(0.25, 0.75)
382         np.random.seed()
383         clean = _sin_f(amp1, 441, srate, n, phase1) + \
384             _sin_f(amp2, 549, srate, n, phase2) + \
385             _sin_f(amp3, 660, srate, n, phase3) + \
386             _sin_f(amp4, 881, srate, n, phase4)
387     else:
388         n = np.tile(np.linspace(0, framelen-1, framelen), (batchsize, 1))
389         phase1 = np.tile(
390             np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
391         phase2 = np.tile(
392             np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
393         phase3 = np.tile(
394             np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
395         phase4 = np.tile(
396             np.random.uniform(0.0, 2*np.pi, batchsize), (framelen, 1)).transpose()
397         amp1 = np.tile(
398             np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
399         amp2 = np.tile(
400             np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
```

```python
        amp3 = np.tile(
            np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
        amp4 = np.tile(
            np.random.uniform(0.25, 0.75, batchsize), (framelen, 1)).transpose()
        # clean = amp * np.sin(2 * np.pi * f / srate * n + phase)
        clean = _sin_f(amp1, 441, srate, n, phase1) + \
            _sin_f(amp2, 549, srate, n, phase2) + \
            _sin_f(amp3, 660, srate, n, phase3) + \
            _sin_f(amp4, 881, srate, n, phase4)
        clean[0:int(batchsize*nop), :] = 0


    def _noise_var(clean, snr_db):
        # we use one noise variance per minibatch
        avg_energy = np.sum(clean*clean)/clean.size
        snr_lin = 10**(snr_db/10)
        noise_var = avg_energy / snr_lin
        print '\tnoise variance for minibatch: ', noise_var
        return noise_var


    # corrupt with gaussian noise
    # use only the signal examples do determine noise variance (in both cases)
    if not sample:
        noise_var = _noise_var(clean[int(batchsize*nop):, :], SIMULATION_SNR)
    else:
        noise_var = _noise_var(clean[int(batchsize*nop):], SIMULATION_SNR)
    noise = np.random.normal(0, noise_var, clean.shape)
    noisy = clean + noise
```

```python
428
429         if sample:
430             noisy = np.array(
431                 [noisy[i:i+framelen] for i in xrange(0, len(noisy), int(pct*framelen))
432             clean = np.array(
433                 [clean[i:i+framelen] for i in xrange(0, len(clean), int(pct*framelen))
434
435         if not sample:
436             labels = np.ones((batchsize, 1))
437             labels[0:int(batchsize*nop)] = 0
438             # labels = np.zeros((batchsize,1))
439             # labels[0:int(batchsize*nop)]=1
440         else:
441             # assuming "noisy" example for sample, not noise example
442             labels = np.ones((batchsize, 1))
443             # labels = np.zeros((batchsize,1))
444         labels = np.tile(labels, (1, framelen))
445
446         return clean.astype(dtype), noisy.astype(dtype), n, labels.astype(dtype)
447
448
449     def stft(x, framelen, overlap=int(pct*framelen)):
450         w = scipy.hanning(framelen)
451         X = np.array([scipy.fft(w*x[i:i+framelen], freq_bins)
452                          for i in range(0, len(x)-framelen, overlap)], dtype=complex64)
453         X = np.transpose(X)
454         return np.abs(X), np.angle(X)
```

```python
455

456

457  def fft(x, fftlen):
458      w = np.tile((scipy.hanning(fftlen)), (batchsize, 1))
459      X = scipy.fft(w*x, fftlen, axis=-1)
460      return np.abs(X).astype(dtype), np.angle(X).astype(dtype)

461

462

463  def gen_freq_data(sample=False, gen_data_fn=gen_data):
464      # for training, use FFTs of any frames
465      # for testing, use FFTs of frames with 25% overlap for proper
466      # reconstruction
467      clean, noisy, n, labels = gen_data_fn(sample)
468      # get FFTs
469      clean_stft = fft(clean, fftlen)   # mag, phase
470      noisy_stft = fft(noisy, fftlen)   # mag, phase
471      return clean_stft, noisy_stft, n, labels  # (mag, phase), (mag, phase)

472

473

474  def istft(X, framelen):
475      frames_avg = int(1/pct)   # 4 in this case
476      # no avg first,
477      overlap = int(pct * framelen)
478      #x = scipy.zeros(int(framelen/2*(time_bins + 1)))
479      x = scipy.zeros(int(X.shape[1]*(X.shape[0]*pct+1-pct)))
480      for n, i in enumerate(range(0, len(x)-framelen, overlap)):
481          x[i:i+framelen] += scipy.real(scipy.ifft(X[n, :]))
```

```python
482        return x


485    def ISTFT(mag, phase, framelen):
486        stft = mag * np.exp(1j*phase)
487        # return np.fft.ifft(stft, framelen)
488        return istft(stft, framelen)


491    def paris_main(params):
492        a, x, s, loss, _, x_hat = paris_net({})
493        train_fn = train(a, x, s, loss)
494        lmse = []
495        predict_fn = theano.function([x], x_hat)

497        np.random.seed(3)
498        clean, noisy, n, _ = gen_freq_data(sample=True)
499        np.random.seed()

501        for i in xrange(params.niter+1):
502            _clean, _noisy, _n, _ = gen_freq_data()
503            loss = train_fn(_noisy[0], _clean[0])
504            LOSSFILE.write(LINEFMT.format(loss))
505            lmse.append(loss)
506            print i, loss

508            if i in range(0, params.niter+50, 50):
```

```
509            # validate mse

510

511            cleaned_up = predict_fn(noisy[0])

512            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))

513            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))

514            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))

515            baseline_mse = mean_squared_error(clean_time, noisy_time)

516            print 'baseline mse:', baseline_mse

517            mse = mean_squared_error(cleaned_up_time, clean_time)

518            print 'mse:', mse

519            MSEFILE.write(LINEFMT.format(mse))

520

521        clean, noisy, n, _ = gen_freq_data(sample=True)

522        cleaned_up = predict_fn(noisy[0])

523        cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))

524        clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))

525        mse = mean_squared_error(cleaned_up_time, clean_time)

526        # print 'mse ', mse

527        wavwrite(normalize(cleaned_up_time),

528                'paris/xhat.wav', fs=srate, enc='pcm16')

529        wavwrite(normalize(clean_time), 'paris/x.wav', fs=srate, enc='pcm16')

530        noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))

531        wavwrite(normalize(noisy_time), 'paris/n.wav', fs=srate, enc='pcm16')

532        plt.figure()

533        plt.subplot(411)

534        # plt.plot(cleaned_up_time[0:fftlen*2])

535        # plt.plot(clean_time[0:fftlen*2])
```

```python
536        plt.plot(cleaned_up_time[1000:1250])

537        plt.plot(clean_time[1000:1250])

538        plt.subplot(412)

539        plt.semilogy(lmse)

540        plt.subplot(413)

541        plt.plot(clean[0][0, :])

542        plt.subplot(414)

543        plt.plot(np.unwrap(clean[1][0, :]))

544        plt.savefig('paris/x.svg', format='svg')


def curro_main(params):
548        g_sig, g_sig_for_real, x, s, loss, g_noi_for_real, x_hat, loss_sig, loss_noi =

549            {})

550        train_fn = train(g_sig, x, s, loss)

551        train_sig = theano.function([x], loss_sig.mean())

552        train_noi = theano.function([x], loss_noi.mean())

553        lmse = []

554        lsig = []

555        lnoi = []

556        predict_fn = theano.function(

557            [x], lasagne.layers.get_output(g_sig_for_real, deterministic=True))

558        predict_fn_noi = theano.function(

559            [x], lasagne.layers.get_output(g_noi_for_real, deterministic=True))

560        both = theano.function(

561            [x], lasagne.layers.get_output(g_sig, deterministic=True))

562
```

```
563    np.random.seed(3)
564    clean, noisy, n, labels = gen_freq_data(
565        sample=True, gen_data_fn=gen_batch_half_noisy_half_noise)
566    np.random.seed()
567
568    for i in xrange(params.niter+1):
569        _clean, _noisy, _n, _labels = gen_freq_data(
570            sample=False, gen_data_fn=gen_batch_half_noisy_half_noise)
571        loss = train_fn(_noisy[0], _labels)
572        lmse.append(loss)
573
574        loss1 = train_sig(_noisy[0])
575        lsig.append(loss1)
576
577        loss2 = train_noi(_noisy[0])
578        lnoi.append(loss2)
579
580        print i, loss, loss1, loss2
581        LOSSFILE.write(LINEFMTLOSS.format(loss, loss1, loss2))
582
583        if i in range(0, params.niter+50, 50):
584            # validate mse
585
586            cleaned_up = predict_fn(noisy[0])
587            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
588            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
589            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
```

```python
590            baseline_mse = mean_squared_error(clean_time, noisy_time)
591            print 'baseline mse:', baseline_mse
592            mse = mean_squared_error(cleaned_up_time, clean_time)
593            print 'mse:', mse
594            MSEFILE.write(LINEFMT.format(mse))
595
596        cleaned_up = predict_fn(noisy[0])
597        noisy_reconstructed = predict_fn_noi(noisy[0])
598        both_ffts = both(noisy[0])
599
600        cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
601        clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
602        noisy_reconstructed = normalize(
603            ISTFT(noisy_reconstructed, noisy[1], fftlen))
604        both_time = normalize(ISTFT(both_ffts, noisy[1], fftlen))
605
606        mse = mean_squared_error(cleaned_up_time, clean_time)
607        mse_noi = mean_squared_error(noisy_reconstructed, clean_time)
608        mse_both = mean_squared_error(both_time, clean_time)
609        # print 'baseline mse', mean_squared_error()  TODO: mse
610        print 'mse ', mse
611        print 'mse of noisy half ', mse_noi
612        print 'mse of combined (both) ', mse_both
613        wavwrite(normalize(cleaned_up_time),
614            'curro/xhat.wav', fs=srate, enc='pcm16')
615        wavwrite(normalize(clean_time), 'curro/x.wav', fs=srate, enc='pcm16')
616        wavwrite(normalize(noisy_reconstructed),
```

```python
                    'curro/nxhat.wav', fs=srate, enc='pcm16')
        wavwrite(normalize(both_time), 'curro/both.wav', fs=srate, enc='pcm16')
        plt.figure()
        plt.subplot(511)
        plt.plot(clean_time[0:fftlen*3])
        plt.plot(cleaned_up_time[0:fftlen*3])
        plt.subplot(512)
        plt.semilogy(lmse)
        plt.subplot(513)
        # plt.plot(cleaned_up[0,:])
        plt.semilogy(
            np.abs(np.fft.fft(np.blackman(cleaned_up_time.size)*cleaned_up_time)))
        plt.subplot(514)
        plt.plot(np.unwrap(noisy[1][0, :]))
        plt.subplot(515)
        plt.plot(noisy_reconstructed[0:fftlen*3])
        plt.savefig('curro/x.svg', format='svg')
        plt.figure()
        plt.plot(lsig)
        plt.plot(lnoi)
        plt.legend(['sig', 'noi'])
        plt.savefig('curro/split.svg', format='svg')


if __name__ == "__main__":
    import sys
    import argparse
```

```python
644    parser = argparse.ArgumentParser()
645    parser.add_argument(
646        'net', type=str, help='super, paris, dan, or curro', default='super')
647    parser.add_argument(
648        '-n', '--niter', type=int, help='number of iterations', default=2000)
649    args = parser.parse_args()
650    mapping = {
651        'super': autoencoder,
652        'paris': paris_main,
653        'dan': dan_main,
654        'curro': curro_main,
655    }
656    mapping[args.net](args)
657    LOSSFILE.close()
658    MSEFILE.close()
```