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

In this thesis, we introduce a modified partitioned autoencoder that de-noises audio without access to clean data. Traditional linear time-invariant (LTI) systems rely on power spectral density (PSD) estimates of desired signals and noise signals, which require some knowledge of the ground truth signals. Popular nonlinear approaches in this area include the use of denoising autoencoders, which are one form of artificial neural networks (ANN). The nonlinearity of neural networks provides additional gains over standard LTI models. However, since de-noising autoencoders also require access to clean data and knowledge of the noise corruption process, we build on existing literature for a semi-supervised partitioned autoencoder that de-noises without the clean signals. We compare existing semi-supervised denoising systems as well as canonical supervised de-noising 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. [1] 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. [2]

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 $f(\centerdot)$ at the output of each neuron. The circle nodes in a neural network diagram can be thought

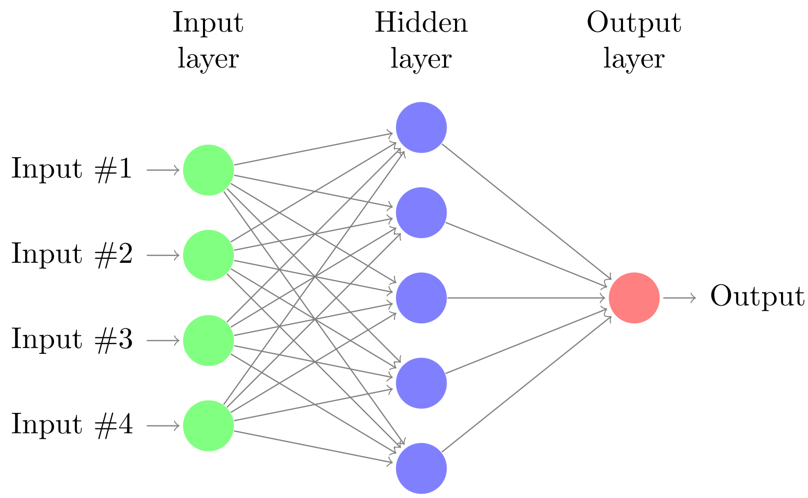

Figure 1: An example neural network. There are 4 input variables, 1 hidden layer with 5 neurons, and 1 output variable. Source: http://www.texample.net/tikz/examples/neural-network/

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

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

We are free to choose activation functions, which we will discuss later. However, note that at the output, the function $g(\cdot)$ is often an identity for regression problems and a sigmoid $\sigma(\cdot)$ 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 $\mathbf{X}$. [2] 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. [1] 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

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

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]

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. [2] 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:

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

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 [1]. 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 [1], the ReLU is subject to a 0-derivative for negative values. The modified ReLU used in [1], which we denote as mReLU, is given by the following:

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

The choice of ϵ used in [1] 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.

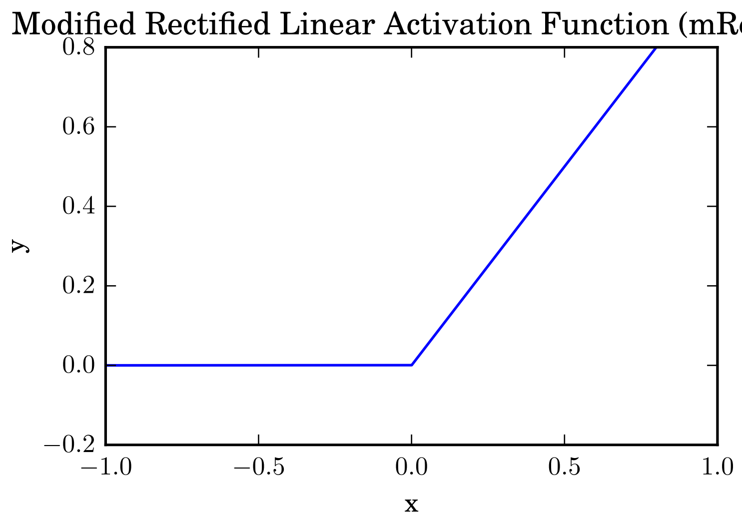

Figure 2: Modified Rectified Linear Unit Activation Function Plot

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 $X[k]$:

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

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

4.2 Partitioned Autoencoder

Adopted from [2], 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 [2], they use the following loss function to accomplish effective partitioning:

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

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 **C** does not change on each training iteration. An example **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 [2], they balance this by choosing 25% of minibatches to be noise-only as well as 25% of latent variables to be noise-only.

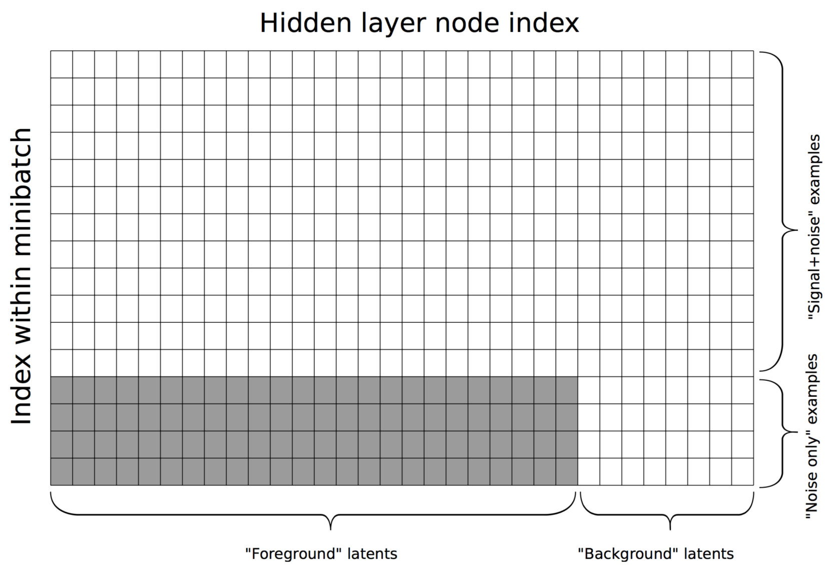

Figure 3: Example Partitioned Masking Matrix. [2] The gray area corresponds to signal (foreground) latents for noise-only examples. We want to penalize the network for any nonzero energy in the signal latents when there are noise-only examples.

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 [2], 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 [1].

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 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 [2], 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.

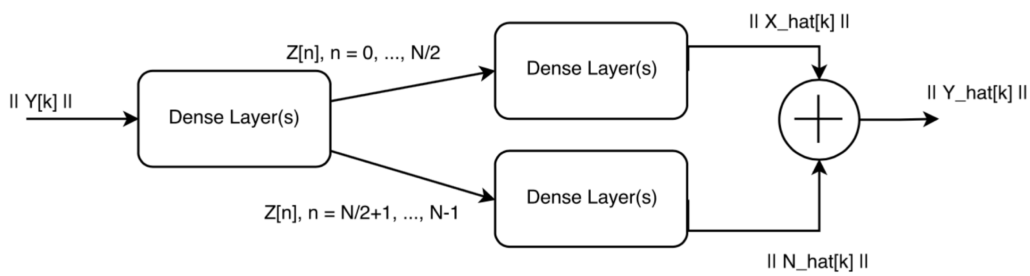

Figure 4: Curro Autoencoder Block Diagram. Partitioning occurs on 50% of the latent space for the signal and the noise. Either can be reconstructed.

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 [2], 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 [2] 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 [2], 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.

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

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

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

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

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.

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

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

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.

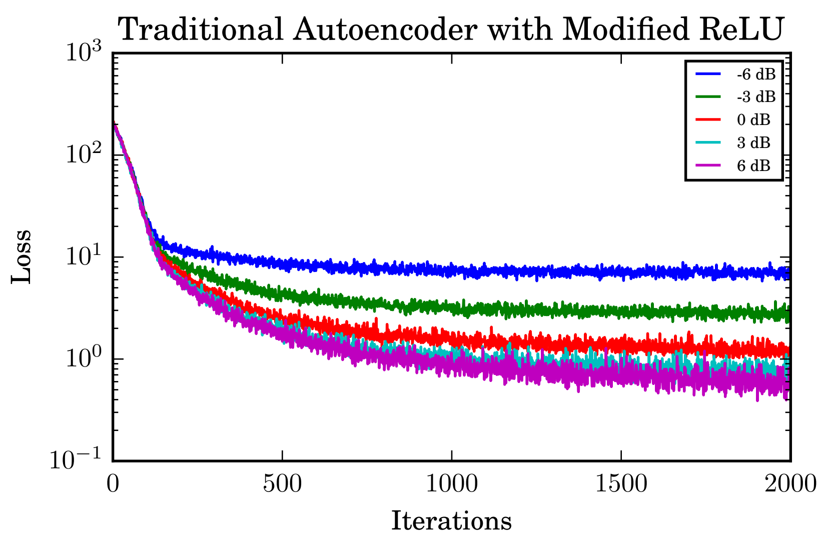

Figure 5: Loss at various SNRs for Supervised Single-Layer Autoencoder with Batch Normalization at the Input

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.

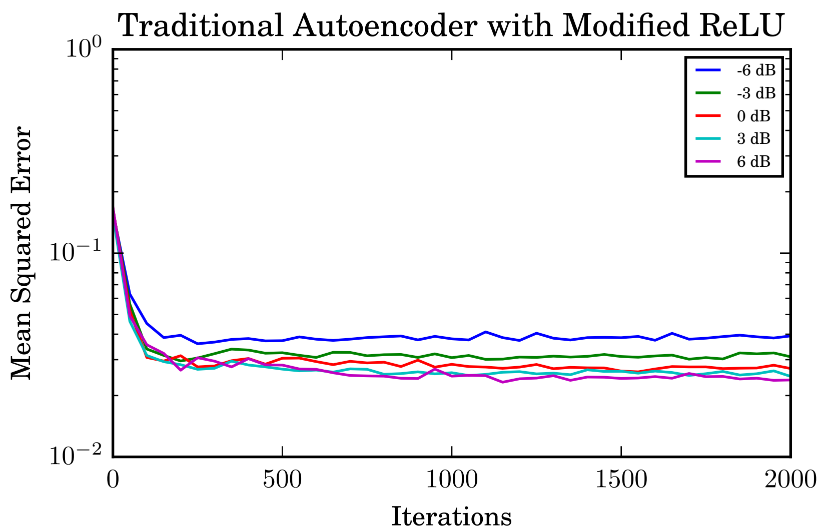

Figure 6: MSE at various SNRs for Supervised Single-Layer Autoencoder with Batch Normalization at the Input

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,

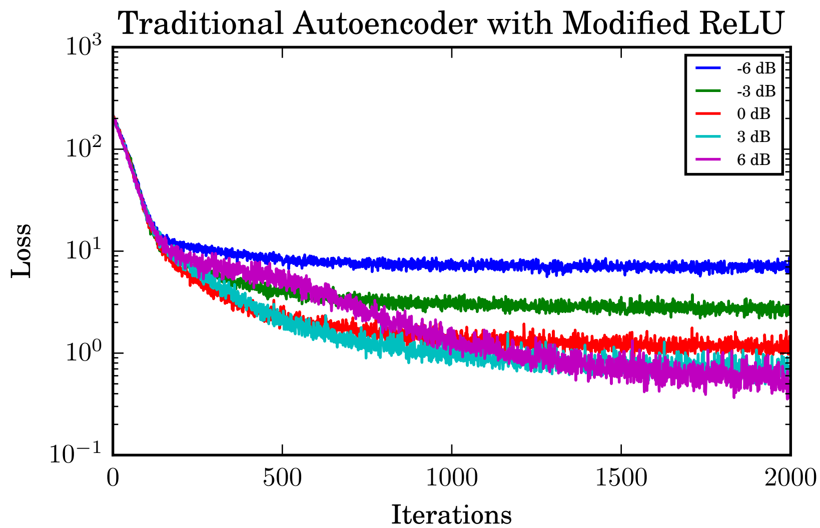

Figure 7: Loss at various SNRs for Supervised Single-Layer Autoencoder without Batch Normalization at the Input

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.

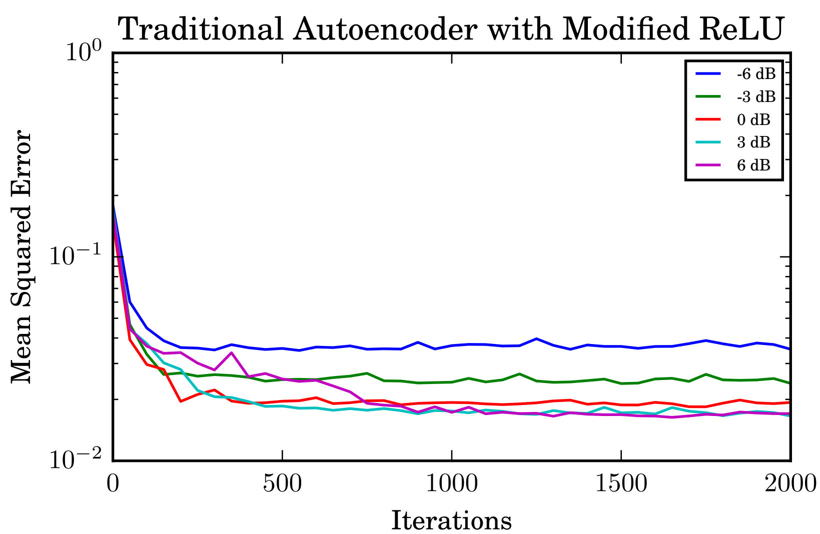

Figure 8: MSE at various SNRs for Supervised Single-Layer Autoencoder without Batch Normalization at the Input

5.2 Partitioned Autoencoder

For the dense partitioned autoencoder, the loss function appears to converge although at a slower rate in Figure 9. A rerun might use more than 2000 iterations. The loss function also converges to a higher magnitude value since the network is not supervised as well as the large regularization coefficients in the loss function.

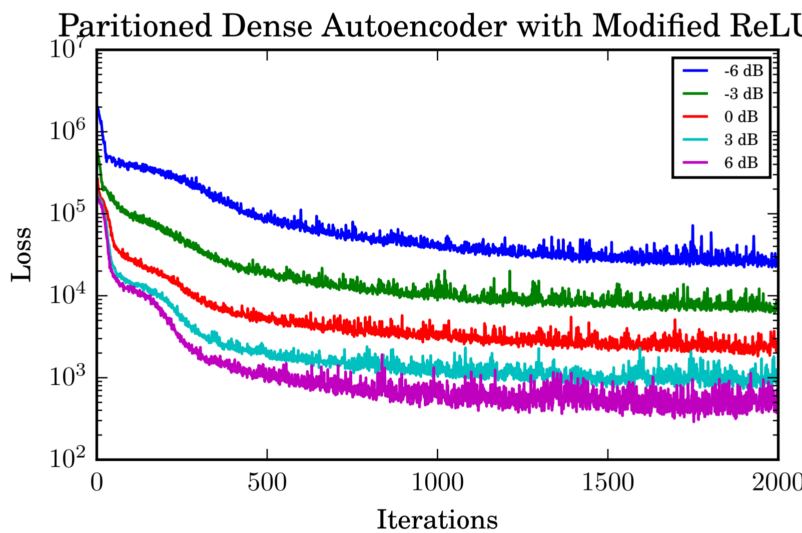

Figure 9: Loss at various SNRs for Single-Layer Partitioned Autoencoder [2]

The MSE is surprisingly low in Figure 10. Unlike in the supervised case, the MSE seems to spread out more as SNR increases. Even at 0 dB, the network seems to learn the noise to some success. A listening test indicates noticeably lower noise level with minimal introduced distortion.

5.3 Partitioned Curro Autoencoder

For the Curro Autoencoder simulation, we used 2 dense layers of size 2048 each before partitioning into two 1024 networks. After the partition, we used two dense layers and an output layer for both partitions, all at size 1024.

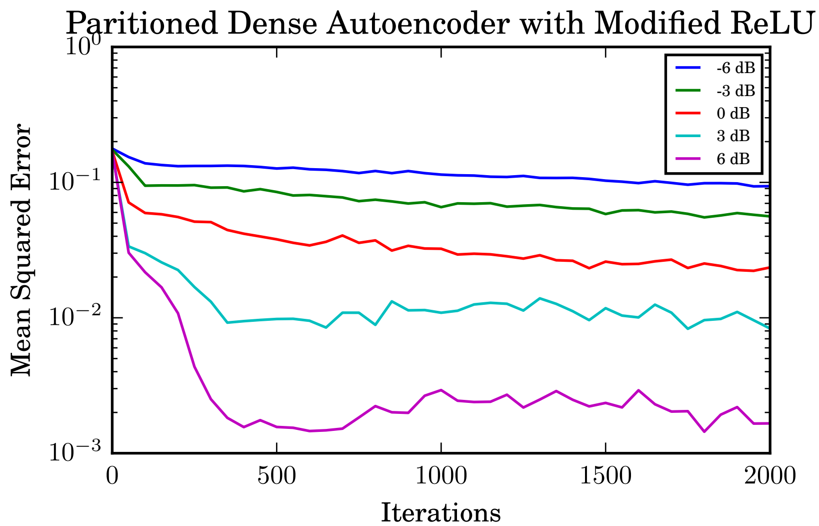

Figure 10: MSE at various SNRs for Single-Layer Partitioned Autoencoder [2]

Interesting to note for the Curro Autoencoder is the fact that it converges almost as quickly across SNR, which can be seen in Figure 11. This suggests that the network might have a lesser dependence on SNR than for the other networks. In terms of making the extension to a real-time system, this suggests that the Curro network outperforms the Dense Partitioned Autoencoder. On the other hand, this could suggest that the network quickly gets stuck in a local minimum and fails to reach lower convergence.

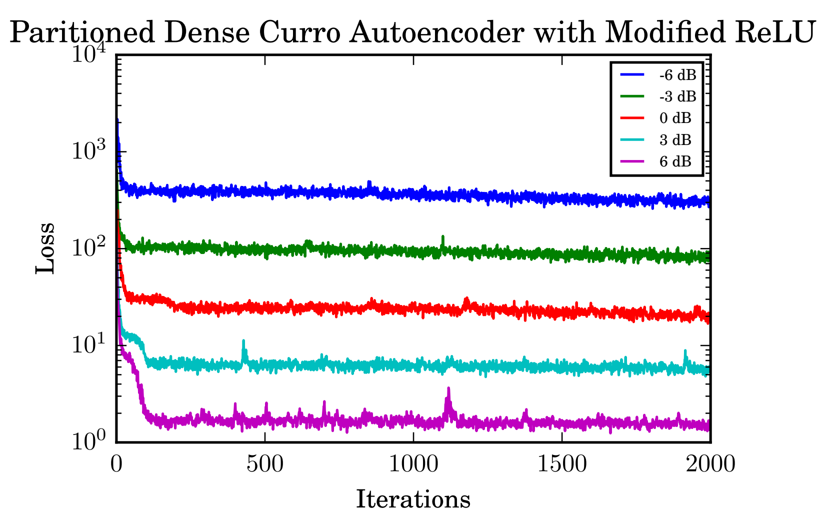

Figure 11: Loss at various SNRs for Single-Layer Curro Autoencoder

In Figure 12, for -6 dB, the time-domain MSE seems to go up slightly after 50-100 iterations, suggesting that either the loss function or the reconstruction could be incorrect. As expected, the network performs better for higher SNR but still seems to get diminishing returns for SNRs greater than 0 dB.

The plot shows the MSE for reconstruction using only the signal half of the network, i.e. the top half. Interestingly, a listening test indicates that summing the two partitions at the output seems to produce a lower noise volume and lower distortion than for either partition individually. This suggests that the network might not be properly partitioning. It is currently unclear as to whether or not this is the result of a bug in the code or a fundamental flaw in the network architecture. Since this network does not have additional dense layers at the summed output, it could be that the network needs to be deeper.

Another theory is that the underlying symmetry in the FFT is causing the network to effectively initialize to two parallel networks. This could be mitigated by only using the first half of the FFT. Also, adding batch normalization to the rest of the layers might result in better convergence and performance.

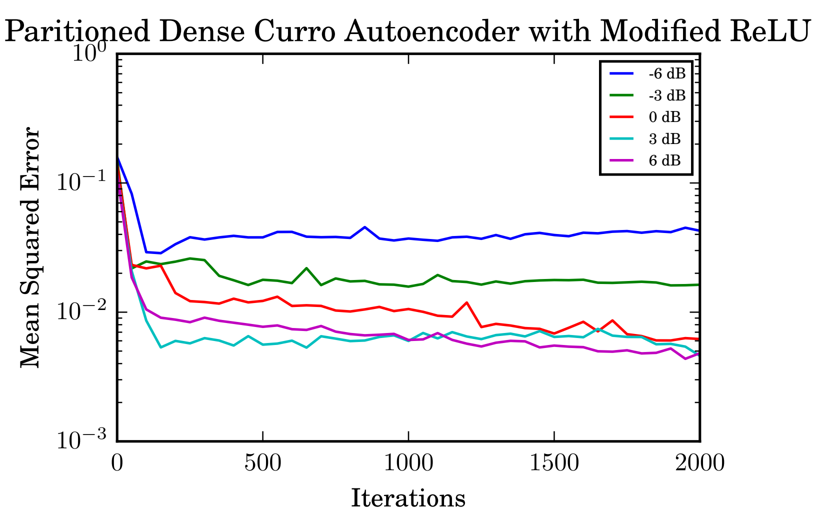

Figure 12: MSE at various SNRs for Single-Layer Curro Autoencoder

5.4 Comparison of Loss Convergence

Since our networks have differing loss functions, it doesn't make sense to compare the converged value. Rather, we would like to compare the networks for convergence time in terms of number of iterations, i.e. number of minibatches exposed to the network. From previous figures, we should expect that the SNR might have some difference depending on the network.

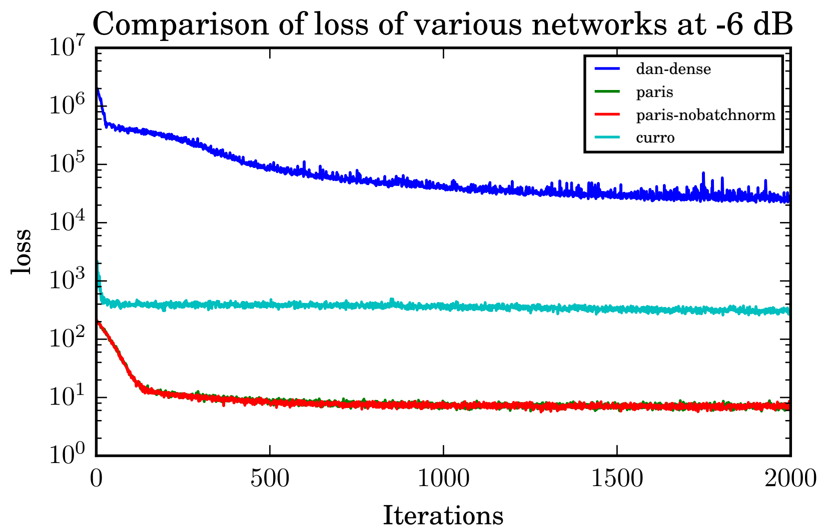

Figure 13: Loss Comparison of Various Networks at -6 dB

In Figures figs. 13 to 17, the Curro Autoencoder seems to converge the quickest, followed by the two supervised networks, then the Dense Partitioned Autoencoder converging the slowest. As expected, as SNR goes up, the supervised networks converge slower while the semi-supervised networks seem to not be as effected by SNR.

5.5 Comparison of Mean Squared Error Convergence

In terms of MSE convergence, we can safely compare both the convergence time and the value of convergence since the reconstruction is based on the same inference example.

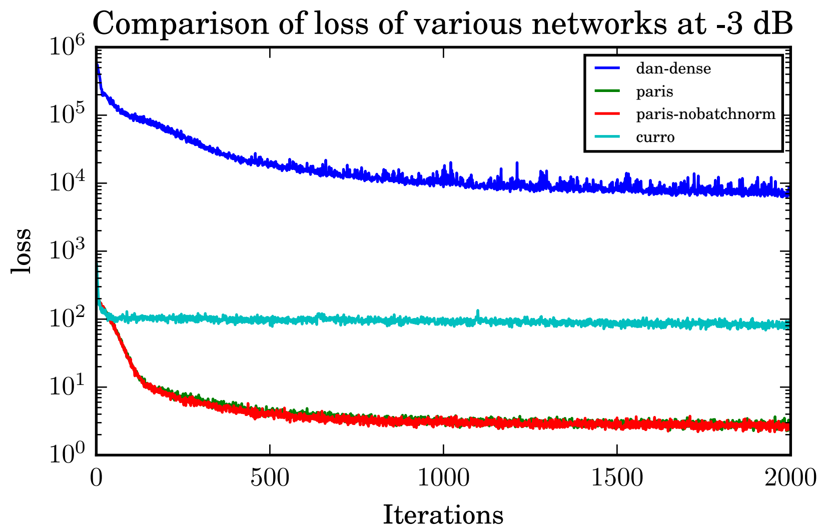

Figure 14: Loss Comparison of Various Networks at -3 dB

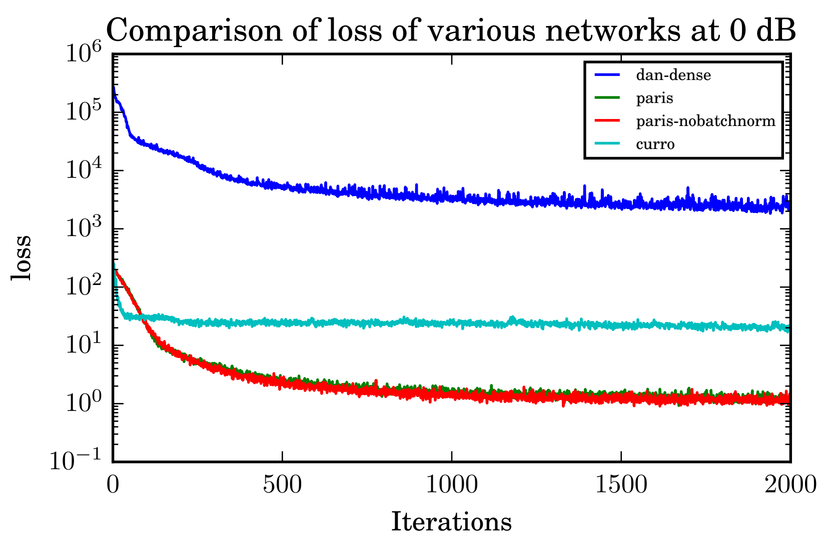

Figure 15: Loss Comparison of Various Networks at 0 dB

In Figures figs. 18 to 20, we see that at low SNR the Dense Partitioned Autoencoder usually has the highest MSE. We expect the supervised autoencoders to have lower MSE, which is the case for these figures. However, the Curro Autoencoder has better performance than the supervised systems which is somewhat unexpected. We should expect that the network which has access

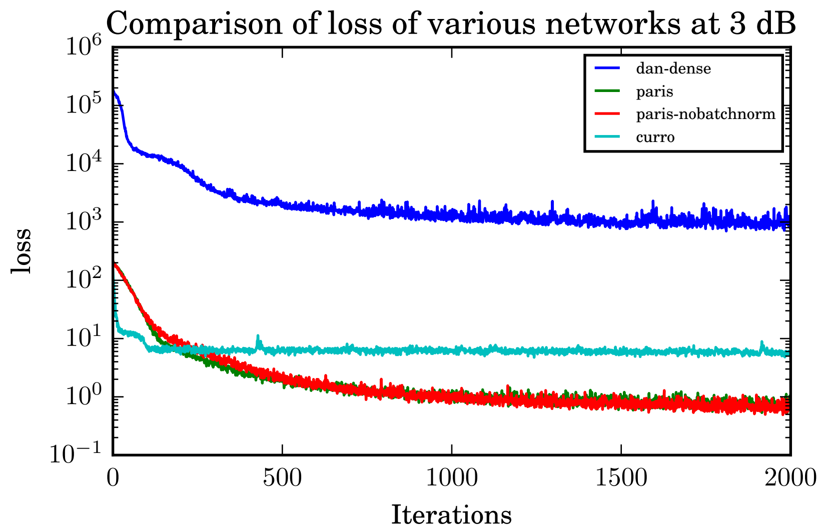

Figure 16: Loss Comparison of Various Networks at 3 dB

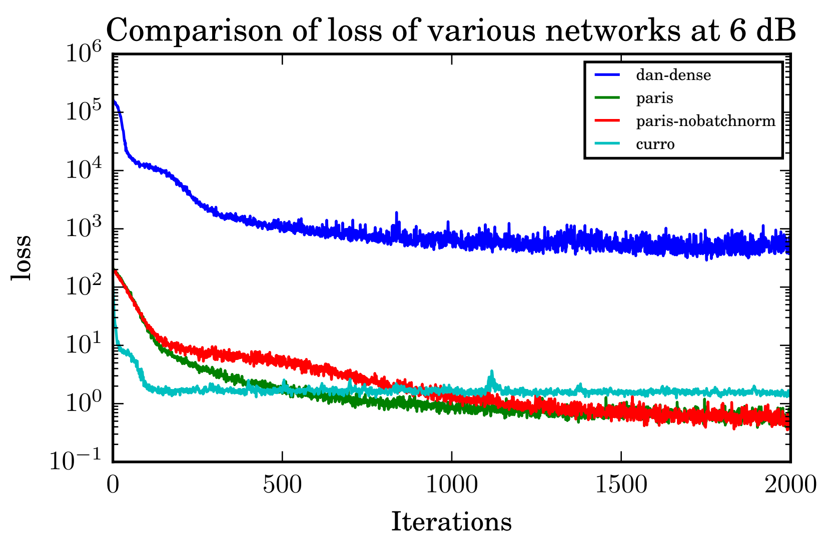

Figure 17: Loss Comparison of Various Networks at 6 dB

to the ground truth during training should converge to a lower MSE. The side effect could be a simulation error, like a bug in the code.

Also interesting to note is that for higher SNR, the Partitioned Dense Autoencoder outperforms both supervised methods. This can be seen in Figures figs. 21 and 22. If the results are correct, this suggests that a semi-supervised network can outperform a supervised network. This might be the result of

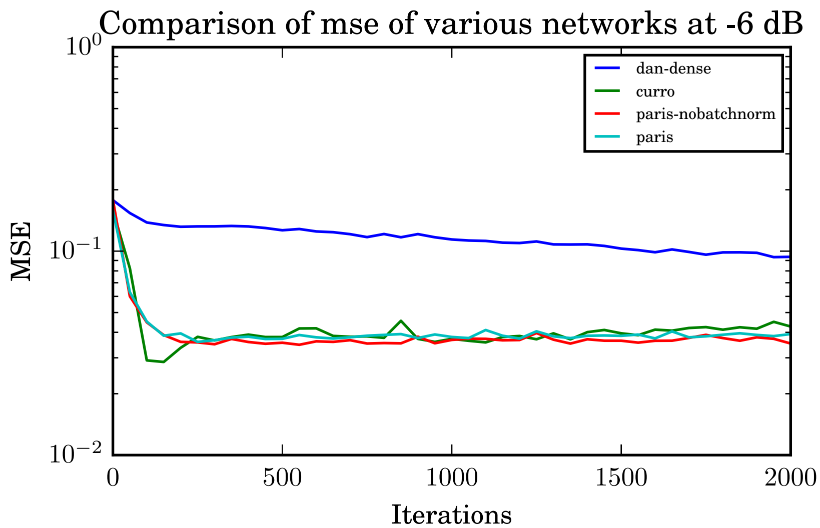

Figure 18: MSE Comparison of Networks at -6 dB

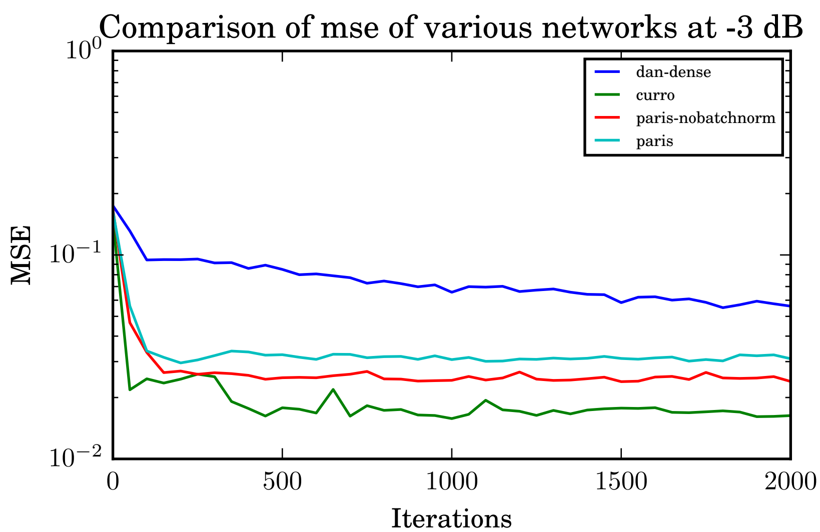

Figure 19: MSE Comparison of Networks at -3 dB

enforcing a structure of the latent space, whereas normally differences in initialization of parameters can cause vastly different latent representations.

Also interesting is the fact that for the highest SNR 6 dB, the Partitioned Dense Autoencoder outperforms the Curro Autoencoder. This can be seen in Figure 22. Again, this could be the result of a simulation error or it could suggest that the Dense Partitioned Autoencoder performs well at high SNR.

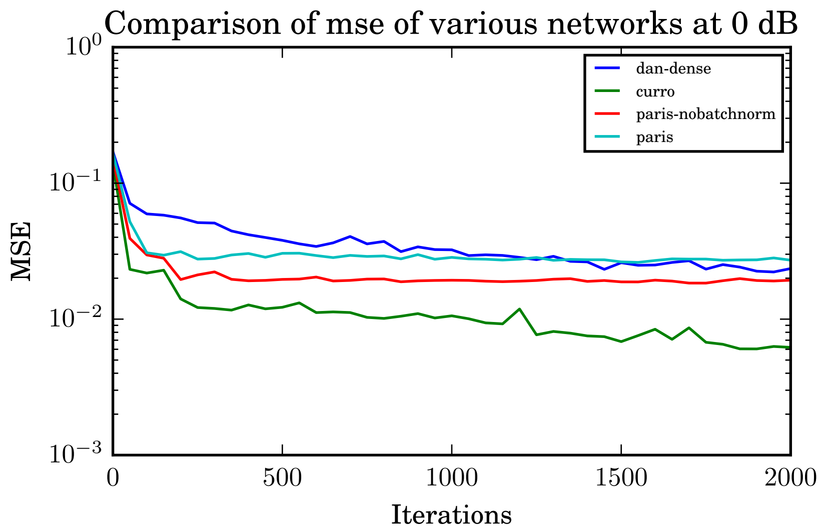

Figure 20: MSE Comparison of Networks at 0 dB

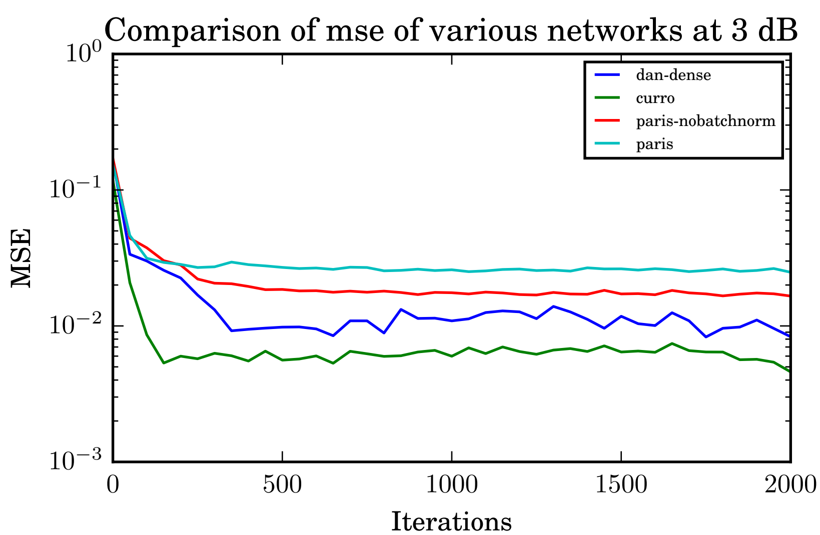

Figure 21: MSE Comparison of Networks at 3 dB

6 Conclusions and Future Work

6.1 Conclusions

Deep partitioned neural network architectures using time and frequency input data seem promising in long-term solutions for denoising speech and music signals. Future work is detailed in the next section.

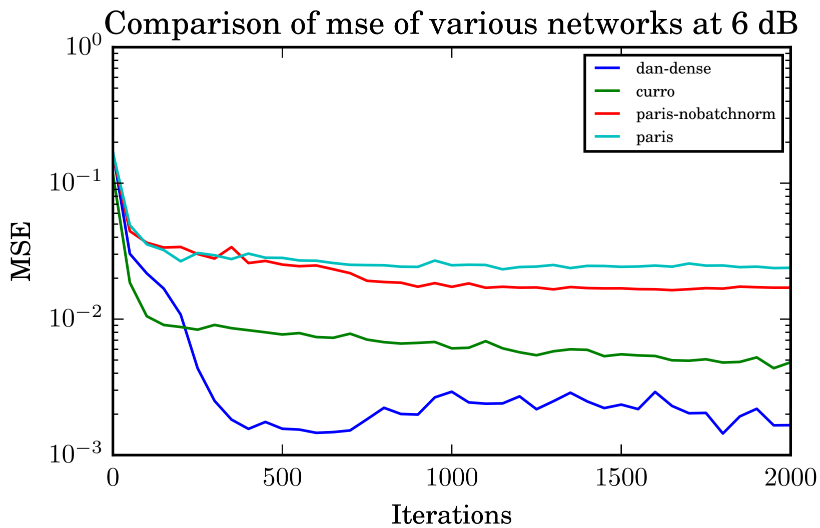

Figure 22: MSE Comparison of Networks at 6 dB

6.2 Future Work

6.2.1 Models

The simulations can easily be extended to multiple hidden layers. These layers can be a combination of convolutional layers as well as fully connected layers. [11] Other recent literature has pointed to recurrent neural networks as an advanced technique for de-noising which has had some success. [12] Batch normalization and layer normalization techniques can help speed up convergence in terms of wall time as well as number of minibatch iterations. [13]

Additionally, the partitions can potentially be constructed to have varying degrees of signal/noise energy such that a more gradual de-noising can occur with less distortion. The partitions can also potentially span more than one layer, which might produce interesting results. Results can then be presented in terms of additional learned parameters which dictate how much of each latent variable to use in reconstruction.

Other metrics could be useful as well in reporting results besides Mean Squared Error. For instance, we could measure the signal-to-distortion ratio (SDR) to identify which networks are introducing distortion and which ones are preventing it. We could also modify our MSE to report as a gain in SNR instead. If we want to preserve audio quality and measure it, we could potentially use user listening tests and audio quality metrics which are based on perceptual models of human hearing.

It's also worth exploring whether or not these models might generalize to similar situations of noisy conditions but with different signals, or vise versa.

6.2.2 Data

Various data sources could be considered in validating the various presented network architectures. Different signal types, e.g. various speech examples and music recordings could provide more useful insights across models and simulations. Similarly, different noise signal types, e.g. restaurant noise, train noise, and crowd noise could provide more insight into how the networks respond.

References

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

[2] 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

[3] C. M. Bishop, *Pattern recognition and machine learning.* springer, 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 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] M. Kayser and V. Zhong, "Denoising convolutional autoencoders for noisy speech recognition."

[12] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks." in *ICML*, vol. 14, 2014, pp. 1764–1772.

[13] J. Lei Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *ArXiv e-prints*, Jul. 2016.

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

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

[24] (2010) Denoising autoencoders (da). [Online]. Available: http://deeplearning.net/dA.html

[25] 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.

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


dtype = theano.config.floatX
batchsize = 128
srate = 16000
pct = 0.5  # overlap
fftlen = 1024
framelen = fftlen

# dan-specific
shape = (batchsize, framelen)
latentsize = 2000
background_latents_factor = 0.25
minibatch_noise_only_factor = 0.5  # also for curro net
n_noise_only_examples = int(minibatch_noise_only_factor * batchsize)
n_background_latents = int(background_latents_factor * latentsize)
lambduh = 0.75

batch_norm = lasagne.layers.batch_norm

```

```python

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


def normalize(x):
    return x / max(abs(x))


def snr_after(x, x_hat):
    return np.var(x)/np.var(x-x_hat)


class ZeroOutBackgroundLatentsLayer(lasagne.layers.Layer):

    def __init__(self, incoming, **kwargs):
        super(ZeroOutBackgroundLatentsLayer, self).__init__(incoming)
        mask = np.ones((batchsize, latentsize))
        mask[:, 0:n_background_latents] = 0
        self.mask = theano.shared(mask, borrow=True)

    def get_output_for(self, input_data, reconstruct=False, **kwargs):
        if reconstruct:
            return self.mask * input_data
        return input_data


def dan_net():
    # net
    x = T.matrix('X')  # input
    y = T.matrix('Y')  # soft label
    network = batch_norm(lasagne.layers.InputLayer(shape, x))
    # network = lasagne.layers.InputLayer(shape, x)
    print network.output_shape
    network = lasagne.layers.DenseLayer(
        network, latentsize, nonlinearity=mod_relu)
    print network.output_shape
    latents = network
    network = ZeroOutBackgroundLatentsLayer(
        latents, background_latents_factor=background_latents_factor)
    network = lasagne.layers.DenseLayer(
        network, shape[1], nonlinearity=lasagne.nonlinearities.rectify)
    print network.output_shape

    # loss
    C = np.zeros((batchsize, latentsize))
    C[0:n_noise_only_examples, n_background_latents + 1:] = 1
    C_mat = theano.shared(np.asarray(C, dtype=dtype), borrow=True)
    mean_C = theano.shared(C.mean(), borrow=True)
    prediction = lasagne.layers.get_output(network)
    mse_term = lasagne.objectives.squared_error(
        prediction, x).sum(axis=[1], keepdims=True)
    scf = lambduh/mean_C
```

```
    regularization_term = scf * y * \
        ((C_mat * lasagne.layers.get_output(latents))
         ** 2).sum(axis=[1], keepdims=True)
    loss = mse_term + regularization_term
    loss = loss.mean()

    # training compilation
    params = lasagne.layers.get_all_params(network, trainable=True)
    updates = lasagne.updates.adam(loss, params)
    train_fn = theano.function([x, y], loss, updates=updates)

    # inference compilation
    predict_fn = theano.function(
        [x], lasagne.layers.get_output(network, deterministic=True, reconstruct=True))

    #
    # other objectives
    #
    square_term = theano.function([x], mse_term.mean())
    regularization_term = theano.function([x, y], regularization_term.mean())

    def do_stuff(network, latents, predict_fn):
        pass

    return network, latents, loss, square_term, regularization_term, train_fn, predict_fn, do_stu


def dan_main(params):
    network, latents, loss, square_loss, reg_loss, train_fn, predict_fn, do_stuff = dan_net()
    lmse = []
    lsq = []
    lreg = []
    # inference example for simulations
    clean, noisy, n, labels = gen_freq_data(
        sample=True, gen_data_fn=gen_batch_half_noisy_half_noise)
    for i in xrange(params.niter+1):
        _clean, _noisy, _n, _labels = gen_freq_data(
            sample=False, gen_data_fn=gen_batch_half_noisy_half_noise)
        # swap 0 and 1 since for dan net, 0 is signal and 1 is background
        _labels = np.expand_dims(np.abs(_labels-1).astype(dtype)[:, 1], axis=1)
        # labels = np.abs(labels-1).astype(dtype)

        loss = train_fn(_noisy[0], _labels)
        lmse.append(loss)

        loss_lsq = square_loss(_noisy[0])
        lsq.append(loss_lsq)

        loss_reg = reg_loss(_noisy[0], _labels)
        lreg.append(loss_reg)
        print '%d\t%.3E\t%.3E\t%.3E' % (i, loss, loss_lsq, loss_reg)

        LOSSFILE.write(LINEFMTLOSS.format(loss, loss_lsq, loss_reg))

```

```
        if i in range(0, params.niter+50, 50):
            # validate mse
            cleaned_up = predict_fn(noisy[0])
            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
            baseline_mse = mean_squared_error(clean_time, noisy_time)
            print 'baseline mse:', baseline_mse
            mse = mean_squared_error(cleaned_up_time, clean_time)
            print 'mse:', mse
            MSEFILE.write(LINEFMT.format(mse))

    cleaned_up = predict_fn(noisy[0])
    print 'freq mse:', mean_squared_error(cleaned_up, clean[0])
    cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
    cleaned_up_clean_phase = normalize(ISTFT(cleaned_up, clean[1], fftlen))
    clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
    noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
    print 'time mse noisy phase:', mean_squared_error(cleaned_up_time, clean_time)
    print 'time mse clean phase:', mean_squared_error(cleaned_up_clean_phase, clean_time)
    print 'baseline time mse noisy to clean:', mean_squared_error(noisy_time, clean_time)
    wavwrite(clean_time, 'dan/x.wav', fs=srate, enc='pcm16')
    wavwrite(noisy_time, 'dan/y.wav', fs=srate, enc='pcm16')
    wavwrite(cleaned_up_time, 'dan/xhat.wav', fs=srate, enc='pcm16')
    wavwrite(cleaned_up_clean_phase,
             'dan/xhat_cleanphase.wav', fs=srate, enc='pcm16')

    plt.figure()
    plt.semilogy(lmse)
    plt.semilogy(lsq)
    plt.semilogy(lreg)
    plt.legend(['overall loss', 'squared error loss', 'regularization loss'])
    plt.savefig('dan/losses.svg', format='svg')


def paris_net(params):
    shape = (batchsize, fftlen)
    x = T.matrix('x')  # dirty
    s = T.matrix('s')  # clean
    #in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))
    in_layer = lasagne.layers.InputLayer(shape, x)
    h1 = batch_norm(
        lasagne.layers.DenseLayer(in_layer, 2000, nonlinearity=mod_relu))
    h1 = lasagne.layers.DenseLayer(
        h1, fftlen, nonlinearity=lasagne.nonlinearities.identity)

    # loss function
    prediction = lasagne.layers.get_output(h1)
    loss = lasagne.objectives.squared_error(prediction, s)
    return h1, x, s, loss.mean(), None, prediction


def curro_net(params):
    # input
```

```python
    shape = (batchsize, framelen)
    x = T.matrix('x')  # dirty input
    label = T.matrix('label')  # noise OR signal/noise

    nonlin = mod_relu

    # 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

    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.identity)

    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, loss_noi
```

```
def autoencoder(params):
    # network
    shape = (batchsize, framelen)
    x = T.matrix('x')  # dirty
    s = T.matrix('s')  # clean
    in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))
    h1 = batch_norm(lasagne.layers.DenseLayer(
        in_layer, 400, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    h2 = batch_norm(lasagne.layers.DenseLayer(
        h1, 330, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    h3 = batch_norm(lasagne.layers.DenseLayer(
        h2, 300, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    h4 = batch_norm(lasagne.layers.DenseLayer(
        h3, 270, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    bottle = h4
    d4 = batch_norm(lasagne.layers.DenseLayer(
        h4, 300, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    d3 = batch_norm(lasagne.layers.DenseLayer(
        d4, 330, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    d2 = batch_norm(lasagne.layers.DenseLayer(
        d3, 400, nonlinearity=lasagne.nonlinearities.leaky_rectify))
    x_hat = batch_norm(lasagne.layers.DenseLayer(
        d2, framelen, nonlinearity=lasagne.nonlinearities.identity))

    # loss function
    prediction = lasagne.layers.get_output(x_hat)
    loss = lasagne.objectives.squared_error(prediction, s)
    reg = 2 * (1e-5 * lasagne.regularization.regularize_network_params(x_hat, lasagne.regularizat
               1e-6 * lasagne.regularization.regularize_network_params(x_hat, lasagne.regularizat
    loss = loss + reg
    return x_hat, x, s, loss.mean(), reg.mean(), prediction


def train(autoencoder, x, s, loss):
    params = lasagne.layers.get_all_params(autoencoder, trainable=True)
    updates = lasagne.updates.adam(loss, params)
    train_fn = theano.function([x, s], loss, updates=updates)
    return train_fn


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

    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

    # f = 440
```

```
    if sample:
        n = np.linspace(0, batchsize * framelen - 1, batchsize * framelen)
        phase1 = np.random.uniform(0.0, 2*np.pi)
        phase2 = np.random.uniform(0.0, 2*np.pi)
        phase3 = np.random.uniform(0.0, 2*np.pi)
        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) + \
        _sin_f(amp4, 881, srate, n, phase4)

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

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

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


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

```

```
    nop = minibatch_noise_only_factor  # noise only percentage of minibatch
    f = 440
    if sample:
        n = np.linspace(0, batchsize * framelen - 1, batchsize * framelen)
        np.random.seed(3)  # to get consistent samples
        phase1 = np.random.uniform(0.0, 2*np.pi)
        phase2 = np.random.uniform(0.0, 2*np.pi)
        phase3 = np.random.uniform(0.0, 2*np.pi)
        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)
        np.random.seed()
        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)
    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) + \
            _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

    if sample:
        noisy = np.array(
            [noisy[i:i+framelen] for i in xrange(0, len(noisy), int(pct*framelen))][0:batchsize])
        clean = np.array(
            [clean[i:i+framelen] for i in xrange(0, len(clean), int(pct*framelen))][0:batchsize])

    if not sample:
        labels = np.ones((batchsize, 1))
        labels[0:int(batchsize*nop)] = 0
        # labels = np.zeros((batchsize,1))
        # labels[0:int(batchsize*nop)]=1
    else:
        # assuming "noisy" example for sample, not noise example
        labels = np.ones((batchsize, 1))
        # labels = np.zeros((batchsize,1))
    labels = np.tile(labels, (1, framelen))

    return clean.astype(dtype), noisy.astype(dtype), n, labels.astype(dtype)


def stft(x, framelen, overlap=int(pct*framelen)):
    w = scipy.hanning(framelen)
    X = np.array([scipy.fft(w*x[i:i+framelen], freq_bins)
                  for i in range(0, len(x)-framelen, overlap)], dtype=complex64)
    X = np.transpose(X)
    return np.abs(X), np.angle(X)


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


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


def istft(X, framelen):
    frames_avg = int(1/pct)  # 4 in this case
    # no avg first,
```

```
    overlap = int(pct * framelen)
    #x = scipy.zeros(int(framelen/2*(time_bins + 1)))
    x = scipy.zeros(int(X.shape[1]*(X.shape[0]*pct+1-pct)))
    for n, i in enumerate(range(0, len(x)-framelen, overlap)):
        x[i:i+framelen] += scipy.real(scipy.ifft(X[n, :]))
    return x


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


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

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

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

        if i in range(0, params.niter+50, 50):
            # validate mse

            cleaned_up = predict_fn(noisy[0])
            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
            baseline_mse = mean_squared_error(clean_time, noisy_time)
            print 'baseline mse:', baseline_mse
            mse = mean_squared_error(cleaned_up_time, clean_time)
            print 'mse:', mse
            MSEFILE.write(LINEFMT.format(mse))

    clean, noisy, n, _ = gen_freq_data(sample=True)
    cleaned_up = predict_fn(noisy[0])
    cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
    clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
    mse = mean_squared_error(cleaned_up_time, clean_time)
    # print 'mse ', mse
    wavwrite(normalize(cleaned_up_time),
             'paris/xhat.wav', fs=srate, enc='pcm16')
    wavwrite(normalize(clean_time), 'paris/x.wav', fs=srate, enc='pcm16')
    noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
```

```
    wavwrite(normalize(noisy_time), 'paris/n.wav', fs=srate, enc='pcm16')
    plt.figure()
    plt.subplot(411)
    # plt.plot(cleaned_up_time[0:fftlen*2])
    # plt.plot(clean_time[0:fftlen*2])
    plt.plot(cleaned_up_time[1000:1250])
    plt.plot(clean_time[1000:1250])
    plt.subplot(412)
    plt.semilogy(lmse)
    plt.subplot(413)
    plt.plot(clean[0][0, :])
    plt.subplot(414)
    plt.plot(np.unwrap(clean[1][0, :]))
    plt.savefig('paris/x.svg', format='svg')


def curro_main(params):
    g_sig, g_sig_for_real, x, s, loss, g_noi_for_real, x_hat, loss_sig, loss_noi = curro_net(
        {})
    train_fn = train(g_sig, x, s, loss)
    train_sig = theano.function([x], loss_sig.mean())
    train_noi = theano.function([x], loss_noi.mean())
    lmse = []
    lsig = []
    lnoi = []
    predict_fn = theano.function(
        [x], lasagne.layers.get_output(g_sig_for_real, deterministic=True))
    predict_fn_noi = theano.function(
        [x], lasagne.layers.get_output(g_noi_for_real, deterministic=True))
    both = theano.function(
        [x], lasagne.layers.get_output(g_sig, deterministic=True))

    np.random.seed(3)
    clean, noisy, n, labels = gen_freq_data(
        sample=True, gen_data_fn=gen_batch_half_noisy_half_noise)
    np.random.seed()

    for i in xrange(params.niter+1):
        _clean, _noisy, _n, _labels = gen_freq_data(
            sample=False, gen_data_fn=gen_batch_half_noisy_half_noise)
        loss = train_fn(_noisy[0], _labels)
        lmse.append(loss)

        loss1 = train_sig(_noisy[0])
        lsig.append(loss1)

        loss2 = train_noi(_noisy[0])
        lnoi.append(loss2)

        print i, loss, loss1, loss2
        LOSSFILE.write(LINEFMTLOSS.format(loss, loss1, loss2))

        if i in range(0, params.niter+50, 50):
            # validate mse
```

```
                cleaned_up = predict_fn(noisy[0])
                cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
                clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
                noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
                baseline_mse = mean_squared_error(clean_time, noisy_time)
                print 'baseline mse:', baseline_mse
                mse = mean_squared_error(cleaned_up_time, clean_time)
                print 'mse:', mse
                MSEFILE.write(LINEFMT.format(mse))

        cleaned_up = predict_fn(noisy[0])
        noisy_reconstructed = predict_fn_noi(noisy[0])
        both_ffts = both(noisy[0])

        cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
        clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
        noisy_reconstructed = normalize(
            ISTFT(noisy_reconstructed, noisy[1], fftlen))
        both_time = normalize(ISTFT(both_ffts, noisy[1], fftlen))

        mse = mean_squared_error(cleaned_up_time, clean_time)
        mse_noi = mean_squared_error(noisy_reconstructed, clean_time)
        mse_both = mean_squared_error(both_time, clean_time)
        # print 'baseline mse', mean_squared_error()  TODO: mse
        print 'mse ', mse
        print 'mse of noisy half ', mse_noi
        print 'mse of combined (both) ', mse_both
        wavwrite(normalize(cleaned_up_time),
                 'curro/xhat.wav', fs=srate, enc='pcm16')
        wavwrite(normalize(clean_time), 'curro/x.wav', fs=srate, enc='pcm16')
        wavwrite(normalize(noisy_reconstructed),
                 '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
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'net', type=str, help='super, paris, dan, or curro', default='super')
    parser.add_argument(
        '-n', '--niter', type=int, help='number of iterations', default=2000)
    args = parser.parse_args()
    mapping = {
        'super': autoencoder,
        'paris': paris_main,
        'dan': dan_main,
        'curro': curro_main,
    }
    mapping[args.net](args)
    LOSSFILE.close()
    MSEFILE.close()
```