# Advance Project Update

Haoyu Fang N18519885

Code:https://github.com/HaoyuCreate/Research-on-Deep-Denoising

## 1 Previous progress

During the 2020 fall semester, I conducted a project of signal denoising and human speech enhancement via an interpretable neural networks. The main goal of the project is to explore how a deep neural network (DNN) performs like a filter in the signal denoising task. Figure 1 illustrates the proposed network's performance of signal denoising on a self-synthesized dataset and Figure 2 demonstrates results of human speech enhancement in a natural scene. The proposed network was coded based on Pytorch [13], one of efficient and popular deep learning platform. However, since Pytorch provides compacted functions to establish a deep network, audiences might have difficulties to observe the inner processes of each computational component. To better understand the mechanism of DNN acting as a filter which removes different patterns of noise (e.g. noise with various amplitudes, background sounds in real scenes and etc.), Prof. Selensnick encouraged me to reproduce the results through a network implemented by third-party-free codes instead of Pytorch [13] or Tensorflow [1]. Therefore, in this updating, I introduce the details of my Numpy-based implementation in Section 2. While implementing the network, I also obverse some interesting facts of the DNN (shown in Section 3) which triggers me to have some new thoughts on improving DNN (discussed in Section 4).


Figure 1: Previous examples of denoised results on a synthesized signal Dataset.

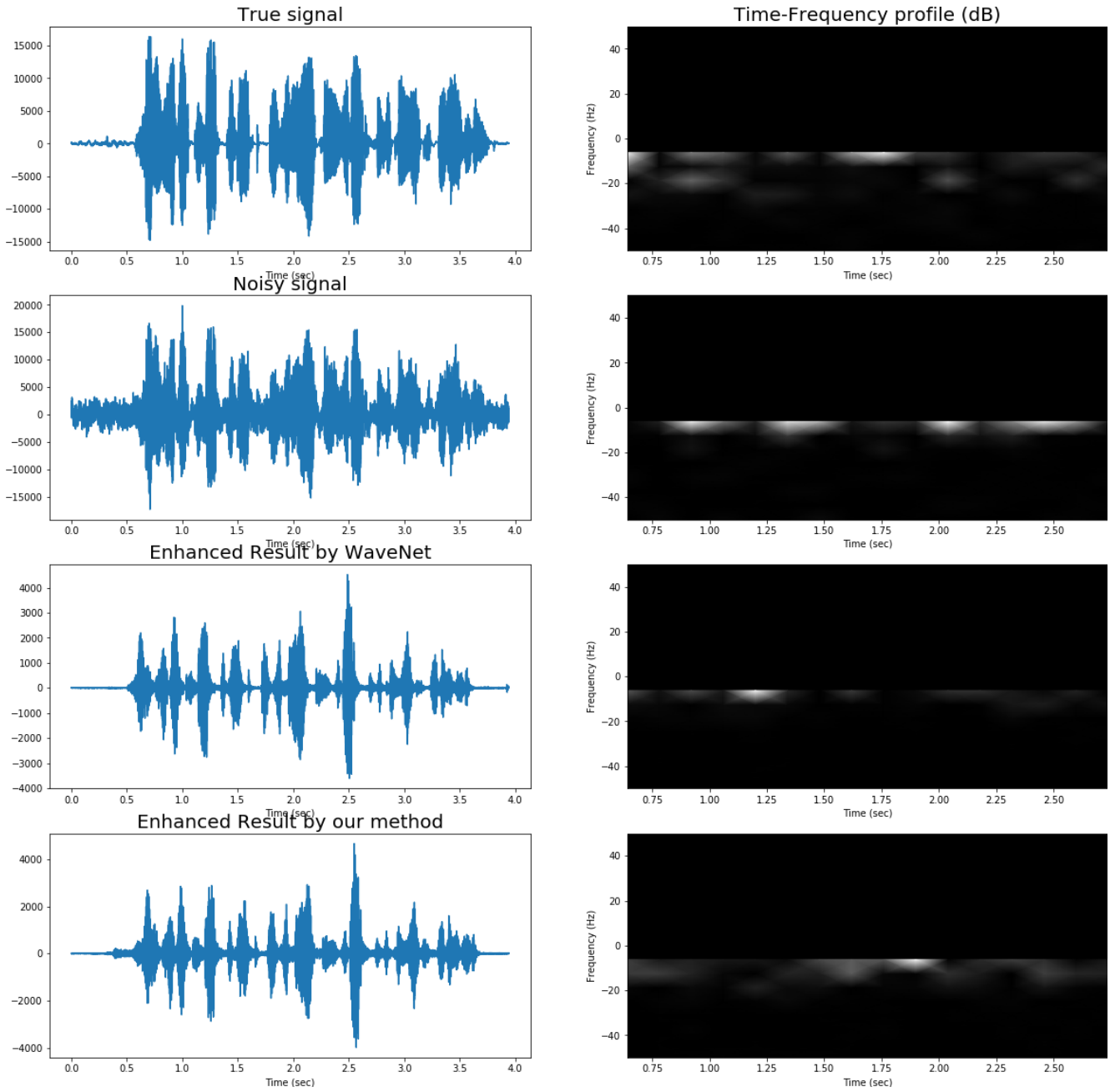


Figure 2: Denoised results on a natural human speech dataset.

# 2 Implementation details

A DNN insists of two branches, one is a forward branch, which handles a 'signal' processing problem; another is a backward branch that solves a optimization issue and find the optimal parameters. When regarding the proposed network as a filter, we are supposed to pay more attention to the learned filters (kernels) rather than feature maps. Therefore, I implement the network's forward branch of the testing process based on Numpy [16] library. I firstly train the previous Pytorch-based implementation to obtain a learned model (weights and bias of each computational layer), and then apply these parameters to current Numpy-based algorithm. In this section, I list some computational components that have significant influence to the final denoising results.

## 2.1 Convolutional layer

One-dimension (1-D) convolution is a vital operation in DNNs. In my Numpy-based implementation, I apply a 1-D convolution over an input signal composed of three input planes. The input has a size of $(N_b, C_{in}, L_{in})$ and the output has a shape of $(N_b, C_{out}, L_{out})$. $N_b$ denotes the number of batches (default is 1 in all my implementation), $C$ is the number of channels and $L$ is the length of the data. Assume the $i$-th kernel has a lenght of $k_i$, and the pad and stride for the convolution operation are $p$ and $s$ respectively. Therefore, the size of weights (a sequence of kernels) and bias of each layer are $(C_{out}, C_{in}, k_i)$ and $(C_{out}, 1)$. For each batch, the output value of the layer can be precisely described as:

$$out(C_{out_i}, j) = bias(C_{out_i}) + \Sigma_{k=0}^{C_{in}-1} weights(C_{out_i}, k) \odot input, \tag{1}$$

where $\odot$ is the valid cross-correlation operator. In my Numpy's implementation, the 1-D convolutional layer realises most functions of its counterpart written based on Pytorch, (e.g. coupling with padding and striding, ability of processing on multiple mini-batches, and etc).

## 2.2 Batch normalization layer

Batch normalization is widely used in DNN designing, it accelerates and stabilizes the DNNs' training through re-centering and re-scaling the inputs of each convolutional layer. I applies batch normalization operation over a 3-D input (a mini-batch of several 1-D inputs) as described:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta, \tag{2}$$

The mean and standard-deviation (SD) are calculated per-dimension over the mini-batches. $\gamma$ and $\beta$ are learnable parameter vectors of size C (where C is the input's channel size). By default, the elements of $\gamma$ are set to 1 and the elements of $\beta$ are set to 0. $\epsilon$ is a small constant to avoid zero denominator. While implementing the batch normalization layer, I find the mean ($E(x)$) and SD ($Var(x)$) have strong influence to the final results even when setting the mini-batch size as one. Note that the means and SDs in Equation 2 are different from the concepts of data's mean and data's SD because they are the statistic information of feature maps. Both of my Numpy-based and Pytorch-based BN layer designs have two modes: 1) using means and SDs globally learned from the entire dataset (fixed parameters); 2) simultaneously calculating the means and SDs of current mini-batch (unfixed parameters). A detailed analysis of the batch normalization layer is introduced in Section 3.3.

## 2.3 Prior learning network

The prior learning network is formed by a Wiener filter and several convolutional layers and it learns deep features from the system function of the signal degradation which is calculated by the Wiener filter. The Wiener filter calculates a rough estimation of the degradation which provides sufficient information about both degradation system and signals. I implement the two Wiener filters via Pytorch and Numpy libraries respectively, which share the same structure. Knowing that signals in our dataset consist of an unknown signal of interest that has been corrupted by additive noises only, the Wiener filter can be simplified as a local-mean filter [9, 17]:

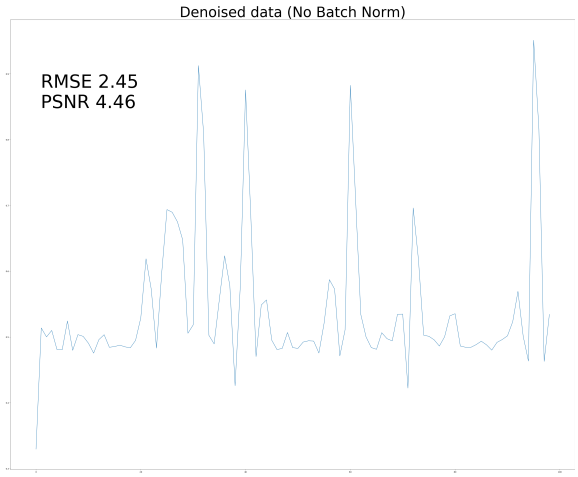


Figure 3: No batch normalization layer (network trained after 200 epochs).

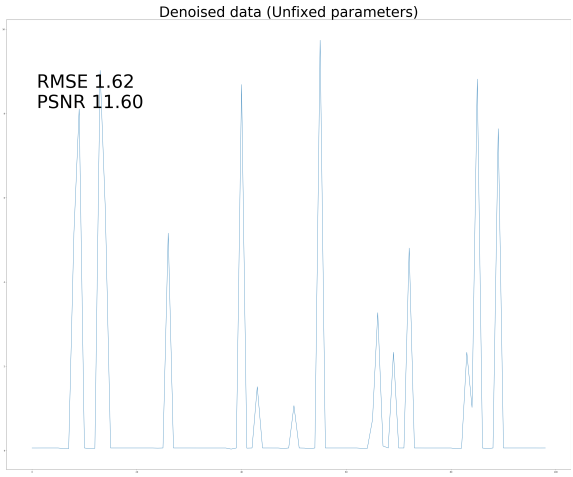


Figure 4: Batch normalization layer with unfixed means and SDs (network trained after 200 epochs).

$$y = \begin{cases} \dfrac{\sigma^2}{\sigma_x^2} m_x + (1 - \dfrac{\sigma^2}{\sigma_x^2}) x = \dfrac{(\sigma_x^2 - \sigma^2)x}{\sigma_x^2} + \dfrac{\sigma^2 m_x}{\sigma_x^2} \\ m_x \end{cases} , \tag{3}$$

where $x$ is the input noisy signal and $y$ is the denoised output. $m_x$ is the local estimate of the mean and $\sigma_x^2$ is the local estimate of the variance. The window for these estimates is an optional input parameter (default is 3). The parameter $\sigma^2$ is a threshold noise parameter. If $\sigma$ is not given then it is estimated as the average of the local variances. Given the input and output signal of the filter, the system function of the degradation can be calculated by $h = \mathscr{F}^{-1}[\mathscr{F}(y)/\mathscr{F}(x)]$, and the inverse system function (denoising system) $h^{-1}$ can be also generated.

# 3 Experimental results and analysis

## 3.1 Comparison between the previously reported model and the current model.

While implementing the network through raw Numpy functions, I keep improving the training process. To avoid trapping in local optimum, a learning rate scheduler with Cosine Annealing [11, 10] strategy is introduced. The scheduler will cut down the learning rate when loss remain the same for a long period but also increase it at a certain rate to skip the local optimum. Another improvement is conducted on gradient scaling. Feature map scaling speeds up gradient descent because weights and biases of kernels will descend quickly on small ranges and slowly on large ranges.

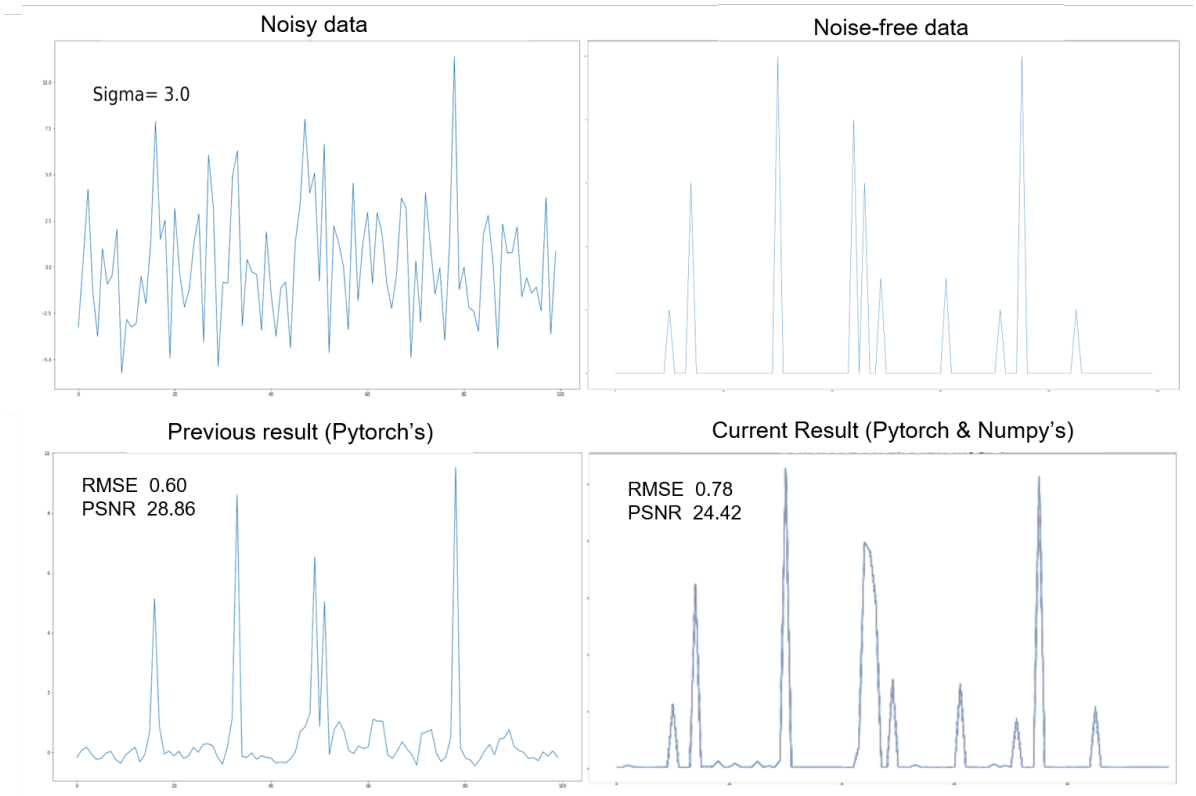


Figure 5: Denoised results through previous and current models.

Compared with denoised examples which were mentioned in my previous report, the current results (both Pytorch and Numpy's version) show inferior performance on some testing signals (shown in **??**) but have an higher average accuracy than the former. There are several reasons resulting in this phenomenon: 1) the new input signal is slightly different from the former since the new input is extracted from a previous image result by an edge detection algorithm [20, 14], which is likely to bring in errors; 2) the previous network is trained with multiple mini-batches at the same time but current network is only fed one mini-batch to be trained or tested for easier observing the weights and biases of the network. Decreasing the batch size could makes the sampling far from the real population which can not ensure a globally optimal weights; 3) the previous results are likely to be caused by a local optimum due to incompleted training. Although the loss of training reached to convergence in that experiment, but it is clear that there are obvious fluctuations in the denoised results and not all noise are removed (shown in Figure 5).

## 3.2 Comparison between Pytorch's and Numpy's implementation of the current model

To reproduce the network with raw Numpy's functions, I rewrite the forward branch of the network and apply an newly learned model including weights and biases to test its performance. Figure 6 shows a comparison between the network based on Pytorch and Numpy libraries. In the figure, it is clear that the denoised results of the two version implementations are identical. Moreover, I also compare the latent feature maps generated by each computational components and find the networks based on two different libraries have the same outputs, which further verifies the correctness of my Numpy's implementation.

## 3.3 Influence of batch normalization layer

Batch normalization (BN) layer is observed to play a important role in denoising tasks. Figure 3 and Figure 4 compare the results of the proposed network without and with batch normalization layers. There is fluctuation in

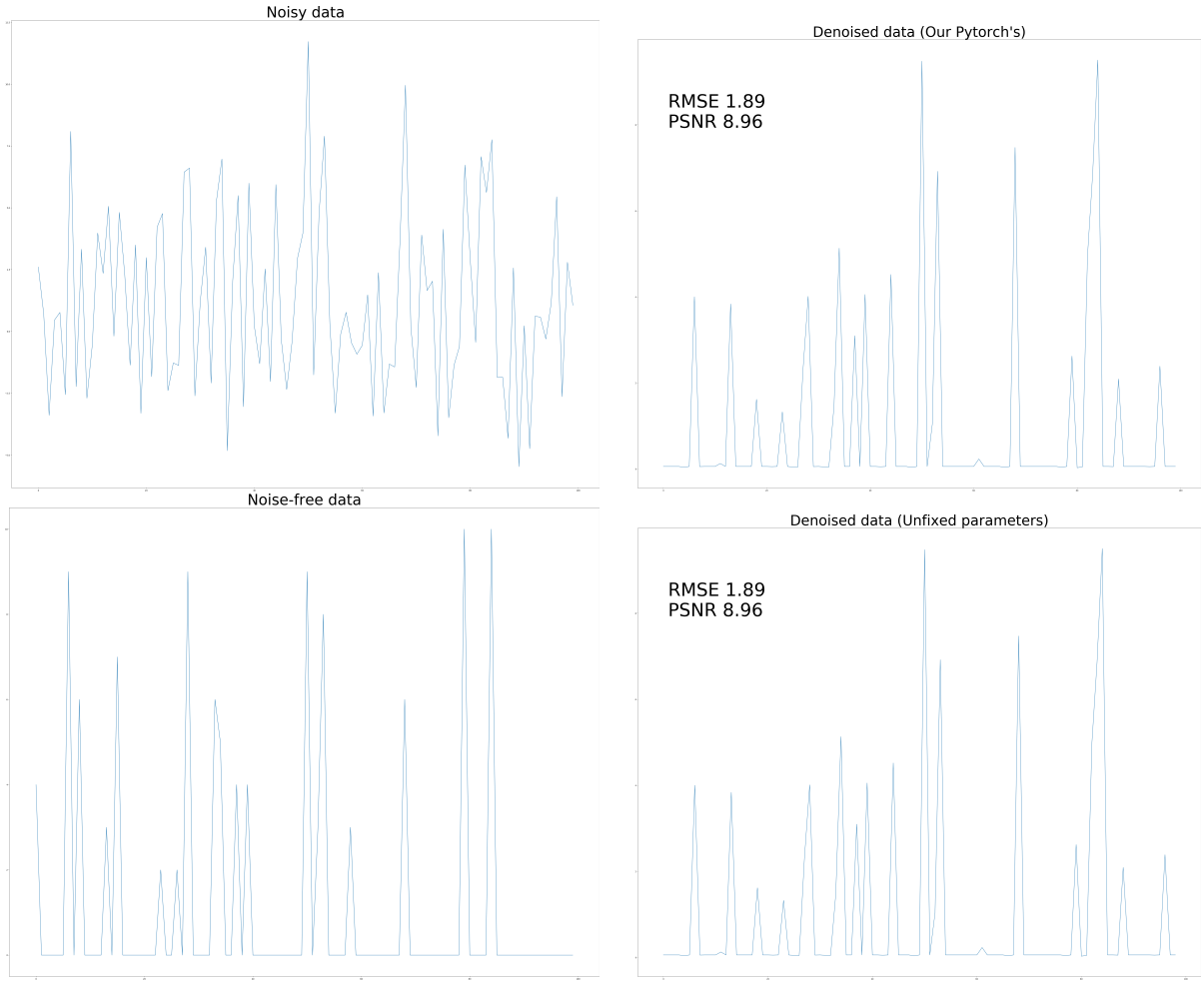


Figure 6: A denoised example, Numpy's and Pytorch's results are identical.

the denoising results which processed by network without batch normalizing operation. The phenomenon is caused by two reasons: 1) the previously well-trained network doesn't complete its training after removing the batch normalization layer; 2) removing the batch normalisation layer also makes the network ignore the the statistic information of the dataset.

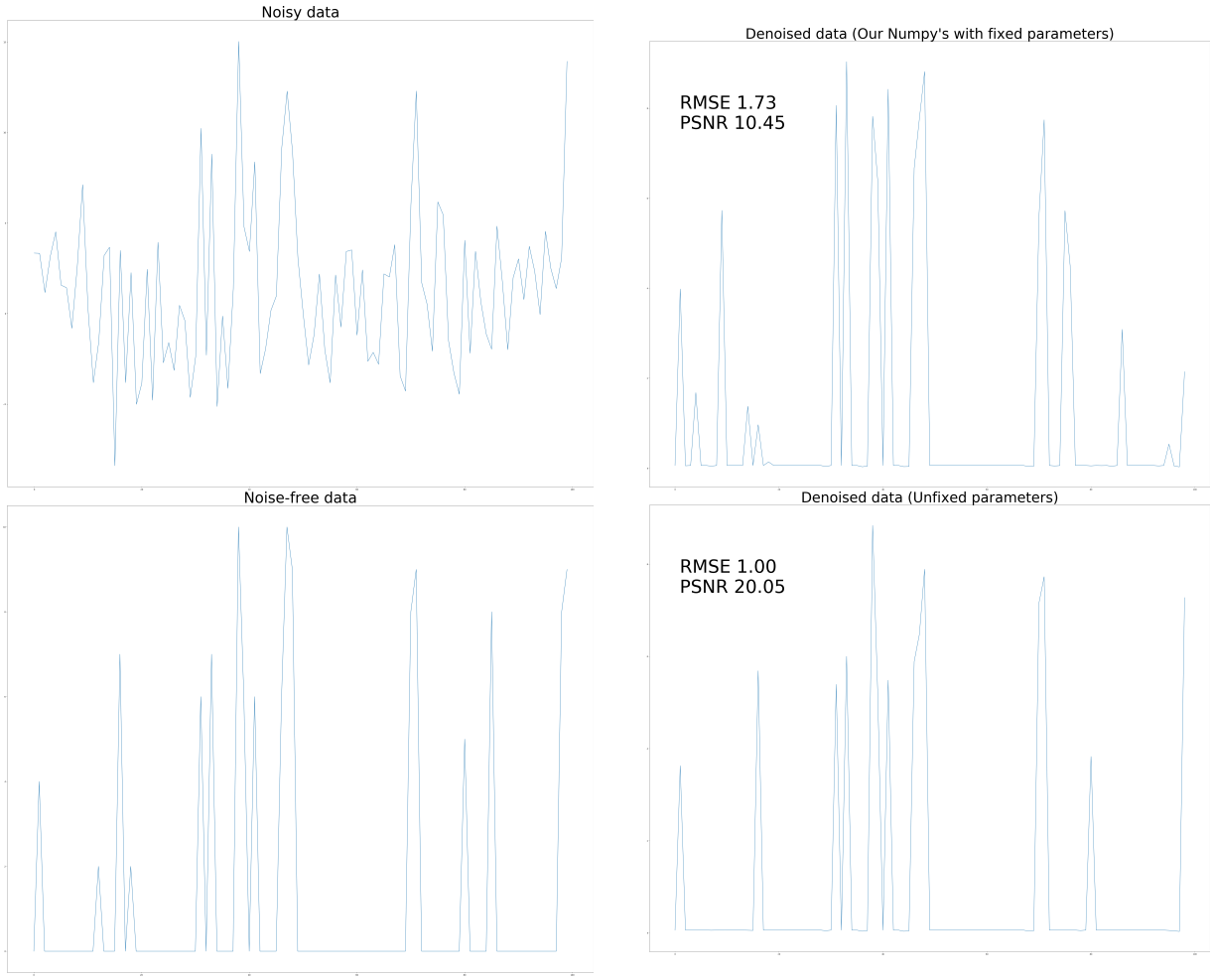


Figure 7: Example #1 between fix parameters and unfixed parameters mode.

In the batch normalization layer, not only the learnable parameters (e.g. $\gamma$ and $\beta$), the statistic knowledge (e.g. means and standard-deviation) of the input feature maps also influence the denoising results. There are many functions to modify statistic-related settings in the Pytorch's library. To change these parameter settings, my Numpy-based implementation also has two modes in terms of the utility of these statistic information. The 'Fixed-parameter mode' refers to a network that uses means and deviations globally learned from the entire dataset, while 'Unfixed-parameter mode' is the network that only uses means and SDs calculated from current mini-batch inputs. Figure 7 shows an extreme example of the denoised results via the networks in different modes. It is observed the network with the unfixed-parameter mode demonstrates stronger capability of removing sparse noises than the fixed-parameter mode. More comparisons of results is introduced in Section 5. However, in most cases, the improvement brought by the unfixed-parameter mode is moderate. In Table 1, the network with the unfixed mode raise its PSNR accuracy by 0.3 and cut down the RMSE loss by 0.003. This is because re-centering or re-scaling the feature maps in the batch normalization layer helps features concentrate on the global means and SDs to avoid the interference of outliers. By this way, the fixed-parameter mode provides a stable testing process but might not output extremely good results for most cases.

Generally, a well-trained network, that involves dropout operation and performs on multiple mini-batches

| Experiment No. | fixed-parameter | unfixed-parameter | wiener filter | L1-norm filter | GM-norm filter [15] |
|---|---|---|---|---|---|
| **PSNR** | | | | | |
| 1 | 9.57 | 9.93 | 4.85 | 4.59 | 1.71 |
| 2 | 9.65 | 9.77 | 4.96 | 4.46 | 1.71 |
| 3 | 9.56 | 9.88 | 4.75 | 4.56 | 1.58 |
| Average | 9.59 | 9.86 | 4.85 | 4.54 | 1.66 |
| **RMSE** | | | | | |
| 1 | 1.84 | 1.80 | 2.40 | 2.44 | 2.88 |
| 2 | 1.83 | 1.82 | 2.39 | 2.45 | 2.88 |
| 3 | 1.84 | 1.80 | 2.41 | 2.44 | 2.90 |
| Average | 1.84 | 1.81 | 2.40 | 2.44 | 2.89 |

Table 1: Average accuracy of the network in different modes and other denoising algorithms tested on 1000 randomly generated noisy signals. Larger PSNR and smaller RMSE reflect better performance.

dataset, is supposed to be tested in the fixed-parameter mode for most tasks. Because the fixed-parameter mode (also named evaluation mode in Pytorch or Tensorflow) sets make all batch normalization layers use globally learned means as well as SDs to guarantee robust and stable outputs. However, in our denoising task, it is reasonable to use unfixed-parameter mode for the testing due to following reasons: 1) our networks do not apply any dropout operations; 2) our network only handles one mini-batch for both testing and training process. 3) my training dataset only have a limited amount of samples, resulting in a phenomenon that population in the dataset can not fully represent the entire data space. For instance, all data in my denoising task, is generated by a Pseudo-random algorithm [2] with $mean = 1.5$ and $var = 1$ but both the training and testing datasets have a slightly different means and deviations due to the limited amount of samples. Therefore, the network uses the means and deviations learned from the input mini-batch to avoid possible errors.

# 4 Analysis and inspiration:

## 4.1 A signal-processing interpretation of supervised DNN-based denoising algorithms

Conventional signal denoising algorithms commonly seek for proper filters (linear or non-linear) to remove the various patterns of noises, while applying different normalization strategies in terms of noise pattern. Equation 4 shows a standard denoising algorithm in time domain, where $x$ denotes clean signal, $y$ is the observed noisy signal, $T(\cdot)$ is a domain transformer, $\odot$ denotes convolution operator and $\Theta(\cdot)$ is a normalization function (e.g. L0-norm, L1-Norm and etc.) The algorithm usually transform input signals into a different feature domain via a feature transformation (e.g. frequency domain, wavelet domain and etc.), where signals of interest can be easier separated from noises. Although denoising signals via DNNs is actually a classification tasks, in which the network identifies each sample in the signal sequence as the signal of interest or the noise. The nature of neural networks make themselves be regarded as deep filters and a good solution to signal denoising tasks.

$$T(x^*) = argmin_x\{\Sigma[T(y) - T(x) \odot k]^2 + \Theta(x)\} \tag{4}$$

The Convolutional layer is similar to domain transformer in traditional methods which put signals into another feature domain, where noises are likely to turn into negative values in feature maps before making the final prediction. ReLU and Leaky ReLU play a vital role in filtering since they can remove features which are under the decision bound. But no of these components can act as a normalization strategy. However, the batch normalization is shown in Equation 5:

$$x^* = \frac{x - MEAN}{\sqrt{VAR + \epsilon}} * \gamma + \beta, \tag{5}$$

where $MEAN$ is a global or a local mean ($E(x)$) and $VAR$ is a global or a local SDs, $\gamma$ and $\beta$ are two learned variables. Compared with Equation 3, the batch normalization layer realise similar functions as a local-mean filter. However, different from the Wiener filter or other norm-based filtering algorithms, batch normalization layer are

not performing on data and only one time, but working on latent features periodically. Since latent features do not have a well-known information (e.g total variance, general total variance and etc.), deep filters like DNNs attempt to normalize the latent features by their own means and variance. In this way, the DNN-based denoising algorithms can be simplified as:

$$\begin{aligned} Training: K^*, \Gamma^*, B^* &= argmin_{K,\Gamma,B}\{\mathscr{L}(Y - \Omega(X \odot K)) + [\Gamma||\frac{X - E(X)}{\sqrt{Var(X)}} - X|| + B]\}, \\ Testing: X^* &= argmin_X\{\Omega(X \odot K) + [\Gamma||\frac{X - E(X)}{\sqrt{Var(X)}} - X|| + B]\}, \end{aligned} \tag{6}$$

where $X = \{x_{in}, x_1, ..., x_{n-2}, x_{out}\}$ is feature maps including input signals and $Y = \{y_{label}, y_{n-1}, ..., y_1\}$ is the labels and latent feature maps, $K = \{k_1, k_2, ..., k_n\}$ denotes a sequence of learnable convolutional kernels. $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$ and $B = \{\beta_1, \beta_2, ..., \beta_n\}$ are learnable constants in each normalization operation. $\Omega(\cdot)$ refers to a strategy of combining feature maps, which does not change in a given network. $\mathscr{L}(\cdot)$ refers to a proper loss function (e.g. negative log likelihood, cross entropy, KL divergence and etc). $||\cdot||$ denotes the activation functions like ReLU or Leaky ReLU. Finally, Equation 6 demonstrates the general case of a supervised DNN-based denoising algorithm including both training and testing processes.

The general form of DNN-based denoising algorithms emphasizes two traits: 1) the methods involve label's information; 2) DNN-based methods conduct filtering or thresholding on latent feature representations. These two traits are highly different from conventional denoising methods and contribute to many state-of-the-art DNN-based denoising approaches. Utility of the label knowledge, which is seldom applied in conventional blind or non-blind denoising algorithms, helps the algorithms lean the probability distribution within the dataset so that the algorithms provide predictions approximating the learned distribution. However, many unsupervised//self-supervised DNN methods without the utility of label knowledge also illustrate strong capability of denoising. One factor contributes to their good performance is that these methods transform input signals into a large amount of feature representative domains and organically fuse these feature representations to learn essential knowledge.

## 4.2 Batch normalization layer with adjustable sparse regularization

Though the initial purpose of introducing batch normalization layers is accelerating network training by reducing internal covariate shift [7], all latent feature representations are filtered in the batch normalization layer by re-centering and re-scaling their values, since the batch normalization layer shares very similar structure to a local-mean filter (shown in Equation 3 and Equation 2). In another words, the feature representations are normalized based on $L-2$ norm regularization. However, in spite of simplicity, universally applying the local-mean filters to normalize all latent feature maps is not efficient since feature representations show significant difference among layers. For instance, it is clear shallow feature maps have rich texture information but deep feature maps are relatively sparse. This observation triggers me to think if we can apply different regularization to latent features in terms of their sparsity while avoiding the covariate shift, because [8, 19] prove that the learning of the neural network converges faster if its inputs are sparse. Based on this idea, Equation 6 can be modified as:

$$\begin{aligned} Training: K^*, \Gamma^*, B^*, \Lambda^* &= argmin_{K,\Gamma,B}\{\mathscr{L}(Y - \Omega(X \odot K)) + [\Gamma||\frac{X - E(X)}{\sqrt{Var(X)}} - X|| + B + \Lambda||X||^p]\}, \\ Testing: X^* &= argmin_X\{\Omega(X \odot K) + [\Gamma||\frac{X - E(X)}{\sqrt{Var(X)}} - X|| + B + \Lambda||X||^p]\}, \end{aligned} \tag{7}$$

where $||\cdot||^P$ is an adjustable $L_p$-norm penalty and $p$ varies according to sparsity of the input $X$. $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ are learnable hyper-parameters similar to $\Gamma$ and $B$ in Equation 6. The newly added penalty is trying to make the output latent features as sparse as possible to help network converge fast.

Many researchers and scholars proposed novel ideas to improve batch normalization layer. For instance, [5] and [3] propose two design of batch normalization layer that attempt to filter outliers in the feature maps to maintain the learned features' distribution more stable. Sparse regularization is introduced into modify ReLU layer [6] to remove the unnecessary zeros outputs from the ReLU and make the feature maps more compact. Therefore, it is very promising to apply sparse regularization to batch normalization layer to achieve better learning capability.

Moreover, the signal-processing interpretation of a general DNN-based denoising algorithm provides some theoretic supports to feasibility of this idea, while increasing advance filtering algorithms with adjustable sparse regularization [4, 12, 18] provides good alternatives and make this idea durable.

# 5 Conclusion and future work

In this report, I introduce details about a Numpy implementation of a previously proposed interpretable deep neural network for signal denoising task. I explain how I programmed the 1-D convolutional layer, 1-D batch normalization layer, ReLU, Wiener filter and other network-related functions. Moreover, I compare the results of the Numpy implementation and previous Pytorch-based network, which verifies that both versions are identical. While implementing the network myself, I conclude a general form of DNN-based denoising approaches from a view of signal processing. Moreover, I observe that batch normalization layer shares a similar structure to a local-mean filter, which inspires me to design a more efficient batch normalization layer that is able to avoid internal covariate shift and sparsify input latent feature representations to accelerate convergence simultaneously.

In the future study, I will try to conclude a general form of unsupervised deep neural networks for denoising tasks, which are more close to blind denoising approaches in practice. Besides, many advanced adaptive filtering algorithms with sparse regularization will be tested to take the place of the traditional batch normalization layers and further improve the capability of deep neural networks.

# References


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on computing*, 15(2):364–383, 1986.

[3] Shang-Hua Gao, Qi Han, Duo Li, Pai Peng, Ming-Ming Cheng, and Pai Peng. Representative batch normalization with feature calibration. In *IEEE Conf. Comput. Vis. Pattern Recog*, 2021.

[4] ZAHRA Habibi, HADI Zayyani, and MOHAMMAD Shams Esfand Abadi. A robust subband adaptive filter algorithm for sparse and block-sparse systems identification. *Journal of Systems Engineering and Electronics*, 32(2):487–497, 2021.

[5] András Horváth and Jalal Al-Afandi. Filtered batch normalization. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 6778–6785. IEEE, 2021.

[6] Hidenori Ide and Takio Kurita. Improvement of learning for cnn with relu activation by sparse regularization. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2684–2691. IEEE, 2017.

[7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[8] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[9] Jae S Lim. Two-dimensional signal and image processing. *Englewood Cliffs*, 1990.

[10] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[11] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[12] Yijie Niu and Jiyou Fei. A sparsity-assisted fault diagnosis method based on nonconvex sparse regularization. *IEEE Access*, 9:59027–59037, 2021.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[14] Weibin Rong, Zhanjing Li, Wei Zhang, and Lining Sun. An improved canny edge detection algorithm. In *2014 IEEE international conference on mechatronics and automation*, pages 577–582. IEEE, 2014.

[15] Ivan Selesnick. Sparse regularization via convex analysis. *IEEE Transactions on Signal Processing*, 65(17):4481–4494, 2017.

[16] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[17] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[18] Minghua Wang, Qiang Wang, Jocelyn Chanussot, and Danfeng Hong. l-l hybrid total variation regularization and its applications on hyperspectral image mixed noise removal and compressed sensing. *IEEE Transactions on Geoscience and Remote Sensing*, 2021.

[19] Simon Wiesler and Hermann Ney. A convergence analysis of log-linear training. *Advances in Neural Information Processing Systems*, 24:657–665, 2011.

[20] Djemel Ziou, Salvatore Tabbone, et al. Edge detection techniques-an overview. *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii*, 8:537–559, 1998.

# Appendix: Extra results


Figure 8: Example #2 between fix parameters and unfixed parameters mode, where parameters refer to running means and standard-deviations in batch normalization layers.


Figure 9: Example #3 between fix parameters and unfixed parameters mode, where parameters refer to running means and standard-deviations in batch normalization layers.


Figure 10: Example #4 between fix parameters and unfixed parameters mode, where parameters refer to running means and standard-deviations in batch normalization layers.