CS230

Untrained Image Restoration using Deep Decoder: Evaluation and Improvements

Tyler Sellmayer
Department of Computer Science
Stanford University
tsellmay@stanford.edu

Salman Naqvi
Department of Computer Science
Stanford University
shnaqvi@alumni.stanford.edu

Abstract

We evaluate and suggest improvements to a neural network image model, the Deep Decoder, as recently described in [1], which can represent and generate natural images well, enabling it to solve inverse problems like denoising and inpainting, with state-of-the-art performance and no pre-training. [1] reported the network's performance on synthetic datasets, such as denoising images corrupted with artificial white gaussian noise and inpainting synthetic white masks. We extend their work by conducting a hyperparameter search to optimize the network's performance on real-world denoising problems. We then evaluate its performance on the benchmark dataset described in [2]. We also investigate the effect of the choice of loss functions on the network's performance showing that the perceptually-motivated loss function described in [3] by far outperforms l_2 loss in visual quality and speed on a real-world inpainting problem of removing movie subtitles.

1 Introduction

The deep decoder, described in [1], is a deterministic function, G, that maps a set of parameters, C, to an image, x. [1] shows that when the cardinality of C is much smaller than the number of free parameters in image x, the stored (G, C) pair works as a compressed representation of the image, and that the compression ratio is on par with modern sparse-wavelet-based image compression algorithms such as JPEG2000.

Given that the network can represent natural images well as $G(C)$, it can be used to solve standard inverse problems with a known forward model, f. Our task then is to find the image x given the transformed and noisy / *damaged image*, $\mathbf{x}^*$. $x^* = f(x) + \eta$. We do that by minimizing, say the l_2/MSE loss with respect to the network parameters, C

$$MSELoss(G, \mathbf{C}, \mathbf{x}^*) = ||G(\mathbf{C}) - \mathbf{x}^*||_2^2 \tag{1}$$

Thus the input to the *algorithm* is only the single image to be recovered, $\mathbf{x}^*$. In the case of denoising, this is the noisy image; in our inpainting experiment this is an image with overlaid subtitles that mask the image's original content. There is no prior training data. The algorithm requires that the network be trained afresh for each damaged image.

For the input to the *network*, we tried two separate approaches: random input constructed as a matrix of uniform i.i.d. noise, as done in [1], and a downsampled version of the damaged image $scale_s(\mathbf{x}^*)$, which is an original contribution. The goal output for the network is always the damaged image $\mathbf{x}^*$.

[1] describes the algorithm comprehensively and analyzes it theoretically. In this paper we present a summary of their algorithm and note the extensions we made.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related work

Other untrained neural network architectures have been applied to these same image-restoration problems, such as the deep "hourglass"-shaped network described in [4]. These deep networks are overparameterized, which means they have a higher capacity to fit the noise and need especial techniques like early stopping for optimal performance. Moreover, they are computationally expensive to run and have a much larger hyperparameter space to search, compared to the smaller Deep Decoder network.

3 Dataset and Features

The original Deep Decoder paper only applies their algorithm to synthetically constructed problems, such as images with artificial white gaussian noise and synthetic white masks.

To evaluate the Deep Decoder's performance on denoising, we use the benchmark dataset of 100 cropped images from [2]. This dataset consists of 512×512 photographs captured by real cameras, with a variety of aperture, shutter speed, and ISO settings. The noisy images are captured with high ISO values (e.g., ISO $= 3200$). The "ground truth" images are captured with low ISO values (e.g., ISO=100). Denoising performance is evaluated by computing the PSNR and SSIM values between the ground truth image and the denoised image $\hat{\mathbf{x}} = G(\mathbf{C})$ generated by the Deep Decoder.

We investigate the Deep Decoder's performance on real-world inpainting problems by attempting movie subtitle removal. We implement an auto-subtitle segmenter as a data preprocessing step shown in 1. It involves taking the image gradient using a Sobel filter, followed by a dilation operation and then binarization to "white out" the subtitles.

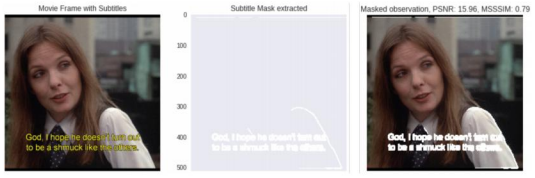

Figure 1: (left to right) Sample frame of [5] with subtitles, extracted mask, and masked image.

4 Method

4.1 Network Architecture

The network consists of L convolution layers with kernel size 1 and channel depth k. After each convolution, we apply a 2x bilinear upsampling transform layer, followed by a nonlinear activation, and then normalization across channels. All the hidden convolutional layers use the same activation function, whereas the output layer uses a Sigmoid function. See 2. [1] describes this network as non-convolutional; this is because the 1x1 kernels do not provide spatial coupling between pixels, which instead arises from the upsampling operation.

For the denoising benchmark, our hyperparameter search found $L = 4$, $k = 196$ and `torch.nn.LeakyReLU(negative_slope=0.1, inplace=True)` [6] to given the best performance.

For inpainting, we used $(k = 256, L = 5)$, which was as close as we could get to the $(k = 320, L = 5)$ used by the original paper [1] within our hardware constraints, along with the same `torch.nn.ReLU()` used in [1].

4.2 Network Input

Before constructing the network input, we calculate its size based on L and the size of the goal image:

$$Height_{\texttt{networkInput}} = \frac{Height_{goal}}{2^L} \tag{2}$$

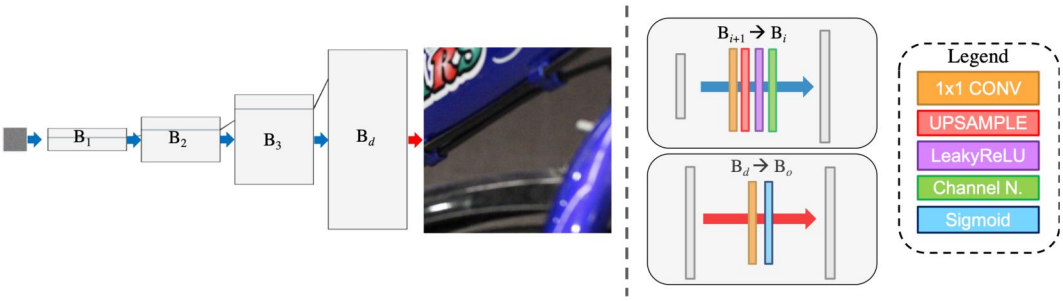

Figure 2: (left) Decoder network architecture with 4 hidden layers. (right) Two distinct sets of operations are shown, one for hidden layers (blue arrows) and one for the output layer (red arrow).

and similarly for $Width_{\texttt{networkInput}}$. When using random noise as the network's input, we generate a $Height_{\texttt{networkInput}} \times Width_{\texttt{networkInput}}$ noise matrix using numpy [7]. When using the downsampled goal image as the network image, we use `PIL.Image.resize` to resize the image to the appropriate dimensions.

Between each iteration of training, a small amount of noise is added to this input for regularization. The regularization noise scaling value σ_r was one of the hyperparameters we determined via hyperparameter search.

4.3 Loss function

The original paper [1] used only mean-squared error (MSE) loss to update the model parameters. We extended their result by experimenting with l_1, l_2/MSE, structural similarity (SSIM) [8] and multi-scale structural similarity (MSSSIM) [9] loss functions.

$$l_1\, Loss(G, \mathbf{C}, \mathbf{x}^*) = |G(\mathbf{C}) - \mathbf{x}^*| \tag{3}$$

$$SSIMLoss(G, \mathbf{C}, \mathbf{x}^*) = \frac{1}{d}\sum_{i=1}^{d} SSIM(x_i, y_i) \tag{4}$$

where x_i and y_i are the $i^{th} 11 \times 11$ local sliding windows over $\mathbf{x}^*$ and $G(\mathbf{C})$, respectively, d is the number of local windows per image, and

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{5}$$

with μ and σ representing the mean and variance of the pixel luminescence values in the window, respectively, and $c_1 = 0.0001, c_2 = 0.0009$ chosen to normalize for the maximum pixel value (e.g., 255) in the representation.

The fourth loss function, $MSSSIMLoss$ [9], does not have a concise formula. It can be described briefly as taking a series of progressively scaled down copies of x and y and calculating their $SSIM$ scores, then linearly combining this set of single-scale scores to give a final multi-scale score.

During our early experimentation with denoising, we evaluated $SSIMLoss$ and linear combinations of $SSIMLoss$ and $MSELoss$ as potential loss functions for training. We trained networks with identical architectures, differing only in their loss functions, and compared the results. Adding $SSIMLoss$ did not cause any improvement on the final PSNR score or SSIM score of the recovered image compared to the ground truth. This is not a surprising result; minimizing MSE trivially maximizes PSNR.

4.4 Training

A single iteration of the training process feeds `networkInput` into the network to generate an output image $G(\mathbf{C}$. We then run backpropagation with the Adam optimizer [10] with learning rate $\alpha = 0.01$ to fit the weights $\mathbf{C}$ of the network G with the goal of minimizing the loss function $L(G, C, \mathbf{x}^*)$. After each iteration, we add normally-distributed regularization noise, scaled by factor σ_r, to `networkInput`.

4.5 Output

After training, the output of the network given the input is the reconstructed image $\hat{\mathbf{x}} = G(\mathbf{C})$. The set of weights $\mathbf{W}$ and the original unregularized network input $network_input$ form the parameters C. The architecture of the network forms the deterministic function G. Thus, the stored values (G, C) form a compressed lossy representation of the damaged image, $\hat{\mathbf{x}} = G(\mathbf{C})$, as described in [1].

4.6 Hyperparameter Search

We implemented a search over the space of hyperparameters described below, to determine a set that maximizes the PSNR score of the network's denoised output compared to the ground truth. The search ran 100 randomized trials of 1000 iterations (ref. 4.4) each, constructing a new network with the chosen hyperparameters and denoising the benchmark image `Canon600D_4-5_125_1600_toy_16_real.JPG`.

- Number of channels in convolutions, k. In initial experiments, this was chosen uniformly from the set [16, 32, 64, 128, 196]. For the final search it was chosen from the set [128, 196], after finding that networks with smaller k were strictly worse.
- Number of convolutional layers, L. In initial experiments, this was chosen uniformly from the set [2, 3, 4, 5, 6, 7]. For the final hyperparameter search we reduced the choices to [3, 4, 5, 6], after finding that networks with 2 or 7 layers were strictly worse.
- Scale of the regularization noise added to the network input between training iterations, η. This was chosen as a logarithmic random variable, $\eta = 10^r$ with $r \sim uniform(-10, -1)$
- Activation function of the convolutional layers, act_func. This was chosen uniformly from the set [`ReLU, ReLU6, Sigmoid, LeakyReLU(0.1, inplace=True)`].
- Constructing random or downsampled network input. This was chosen uniformly at random.

5 Results and Discussion

5.1 Optimal Denoising Model & Benchmark

Hyperparameter	Value in best-scoring denoiser
Number of channels in 1x1 convolutions, k	196
Number of convolutional layers, L	4
Input regularization noise, σ_r	$1.75e-7$
Activation function of hidden layers	`LeakyReLU(negative_slope=0.1)`
Input type: Random or downsampled	`random`

Table 1: The winning hyperparameters from our hyperparameter search which yield the best denoising performance, i.e. the reconstructed image from this network demonstrated the highest PSNR and SSIM scores compared to the ground truth after 1000 iterations of training.

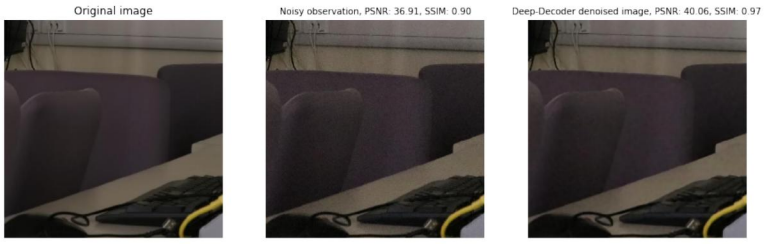

Figure 3: Deep Decoder exhibits state-of-the-art denoising on the real-world benchmark dataset. Zoom in on the PDF to see the effect more closely.

According to Table VII of [2], which reports benchmark results of average PSNR and SSIM scores for existing denoising algorithms, our algorithm's performance is on par with the current state-of-the-art.

[2] reports the scores for 14 algorithms; of these, 8 have higher average PSNR scores than our algorithm, and 7 have higher SSIM scores than our algorithm.

	PSNR	SSIM
Mean	37.1929	0.9558

Table 2: The results of running $10,000$ training iterations of the network with the hyperparameters described above 1 on all 100 images from the benchmark dataset [2]. An example result image is shown in 3.

5.2 Inpainting Performance Enhancement with MSSSIM+L1 loss

The algorithm's performance in inpainting was found to be strongly dependent on the choice of loss function used. In particular, as shown by [3], the linear combination of MSSSIM and l_1 loss functions (MSSSIM+L1) really shines here. As seen in 4, deep decoder converges much faster with MSSSIM+L1 and reaches an acceptable solution within 1000 iterations in contrast to when using MSE.

(a) Inpainting results after 100 iterations.

(b) Inpainting results after 1000 iterations.

Figure 4: After 100 iterations, (left) MSE loss has not achieved good inpainting but (right) mix (MSSSIM+L1) loss has already blurred out the masked region completely. After 1000 iterations, (left) MSE loss still leaves white artifacts, but (right) mix loss has nearly completed.

6 Conclusion / Future Work

We found a new best set of hyperparameters for the Deep Decoder and showed that it performs on par with the state-of-the-art algorithms on a real-world image denoising benchmark. We demonstrated the Deep Decoder's ability to give good inpainting results on a real-world movie subtitles sample images, and showed that MSSSIM+L1 loss speeds up the inpainting process compared to MSE loss. We also investigated the network's application on the image deblurring problem by implementing a Gaussian kernel forward model in PyTorch. Due to lack of time, we leave it for future completion.

Our hyperparameter search was limited by time and resource constraints. Future researchers may explore a larger hyperparameter space, or choose to optimize for speed, by finding hyperparameters that minimize the number of training iterations required to surpass some threshold PSNR and SSIM scores. Our search used only one example image for scoring; future work could conduct a search using the average scores on a larger set of images.

Note that in our experiments, equation (2) imposes a restriction on the depth of the network given the size of the goal image: $L \leq \log_2(\min(Height_{\texttt{goal}}, Width_{\texttt{goal}}))$ Future researchers may perform upsampling differently. One approach would be to fix the network input size at some static value, and add as many upsampling layers as required to reach the desired output size.

Our denoising approach may be useful for reconstructing clean audio by denoising spectrogram images, as an alternative to custom spectrogram-denoising algorithms like the one described in [11].

7 Contributions

Tyler implemented the hyperparameter search described in 4.6, applied the SSIM loss function (4) from [12] to the model, set up AWS, set up a development environment on his personal GPU hardware, and ran the benchmarks described in [2].

Salman investigated the use of optimal loss functions for image restoration problems and implemented the MSSSIM+L1 ("Mix") loss function as described in [3], using the MSSSIM loss function implementation in PyTorch from [13]. He also attempted the movie subtitle removal problem including implementing subtitle segmenter and comparing performance using different loss functions. He further investigated extending deep decoder to addressing deblurring problem by incorporating a Gaussian kernel PyTorch implementation as the forward model but couldn't get results on time. On the logistics side, he converted the PyTorch implementation from [1] into Google CoLab compatible notebooks and use its GPU Runtime.

We both contributed to the final poster and paper.

8 Code

Our code repository can be found on GitHub: https://github.com/tsell/cs230-project. It contains several saved IPython notebooks demonstrating our results:

1. benchmark_with_best_hyperparameters.ipynb holds the hyperparameter search and benchmark functions.
2. benchmark_statistics.ipynb holds the benchmark statistics calculations.

We built on the implementation of Deep Decoder [1] from the original authors, available here: https://github.com/reinhardh/supplement_deep_decoder.

We used an implementation of SSIM in PyTorch from Po-Hsun Su [12], available here: https://github.com/Po-Hsun-Su/pytorch-ssim.

We also used an implementation of MS-SSIM in PyTorch from Jorge Pessoa [13], available here: https://github.com/jorge-pessoa/pytorch-msssim

The benchmark dataset [2] is available here: https://github.com/csjunxu/PolyU-Real-World-Noisy-Images-Dataset.

References

[1] R. Heckel and P. Hand, "Deep decoder: Concise image representations from untrained non-convolutional networks," *CoRR*, vol. abs/1810.03982, 2018.

[2] J. Xu, H. Li, Z. Liang, D. Zhang, and L. Zhang, "Real-world noisy image denoising: A new benchmark," *CoRR*, vol. abs/1804.02603, 2018.

[3] H. Zhao, O. Gallo, I. Frosio, and J. Kautz, "Loss functions for image restoration with neural networks," *IEEE Transactions on Computational Imaging*, vol. 3, no. 1, pp. 47–57, 2017.

[4] D. Ulyanov, A. Vedaldi, and V. S. Lempitsky, "Deep image prior," *CoRR*, vol. abs/1711.10925, 2017.

[5] W. Allen and C. H. Joffe, "Annie Hall," 1977.

[6] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, p. 3, 2013.

[7] O. Travis E, "A guide to NumPy," 2006.

[8] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[9] Z. Wang, E. P. Simoncelli, and A. C. Bovik, "Multiscale structural similarity for image quality assessment," in *The Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2, pp. 1398–1402, Ieee, 2003.

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

[11] A. Mallawaarachchi, S. Ong, M. Chitre, and E. Taylor, "Spectrogram denoising and automated extraction of the fundamental frequency variation of dolphin whistles," *The Journal of the Acoustical Society of America*, vol. 124, no. 2, pp. 1159–1170, 2008.

[12] P.-H. Su, "PyTorch SSIM." `https://github.com/Po-Hsun-Su/pytorch-ssim`, 2017.

[13] J. Pessoa, "PyTorch MS-SSIM." `https://github.com/jorge-pessoa/pytorch-msssim`, 2018.

[14] F. Perez and B. E. Granger, "IPython: A system for interactive scientific computing," *Computing in Science Engineering*, vol. 9, pp. 21–29, May 2007.

[15] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science Engineering*, vol. 9, pp. 90–95, May 2007.

[16] PythonWare, "Python imaging library (PIL)." `http://www.pythonware.com/products/pil/`, 2010–2018.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017.