Fiber bundle image restoration using deep learning

Jianbo Shao

THE UNIVERSITY OF ARIZONA

10/31/2018

Background

Fiber bundle (FB) image is affected by repeated honeycomb patterns and its spatial resolution is limited by core size and density.

Lens tissue sample image.
Core diameter $\approx 14\mu m$.
Sensor pixel size $= 3.45\mu m$.

Background

So we have two goals for improving fiber bundle imaging systems:

- Remove fixed pattern noises.
- Enhance spatial resolution.

Review of current technology

Existing methods can be categorized into two groups, based on whether a single FB image is used for input or multiple FB images are used for input.

Single image method

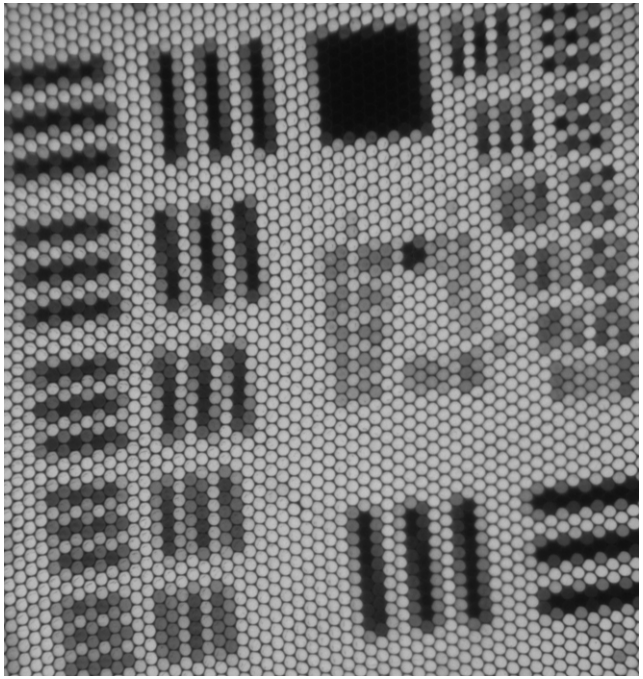
- Fourier domain filtering.
- Spatial domain filtering.
- Interpolation.

Multiple images method

- Create median image by using knowing motions.
- Average image by randomly transversely shifting the probe.

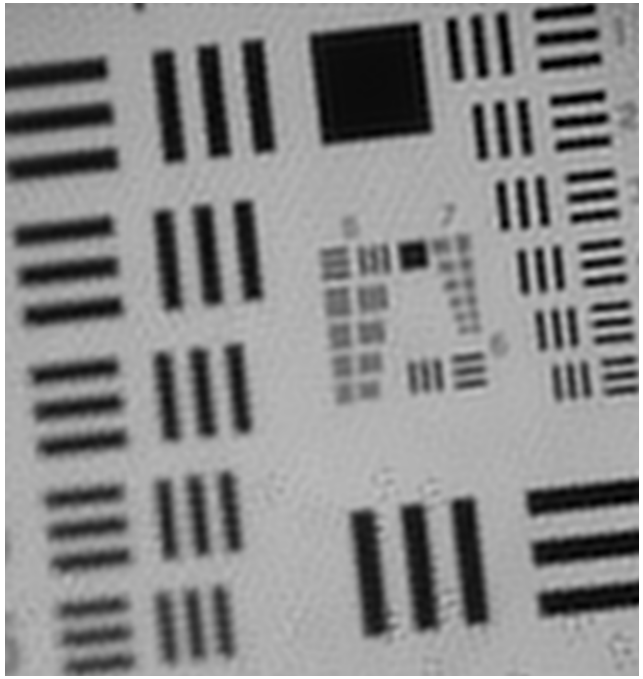
Previous proposed method

We previously proposed a MAP based multi-frame FB resolution enhancement method [1].



16 LR fiber bundle images

Reconstruction



HR image

1. Jianbo Shao, Wei-Chen Liao, Rongguang Liang, and Kobus Barnard, "Resolution enhancement for fiber bundle imaging using maximum a posteriori estimation," Opt. Lett. 43, 1906-1909 (2018)

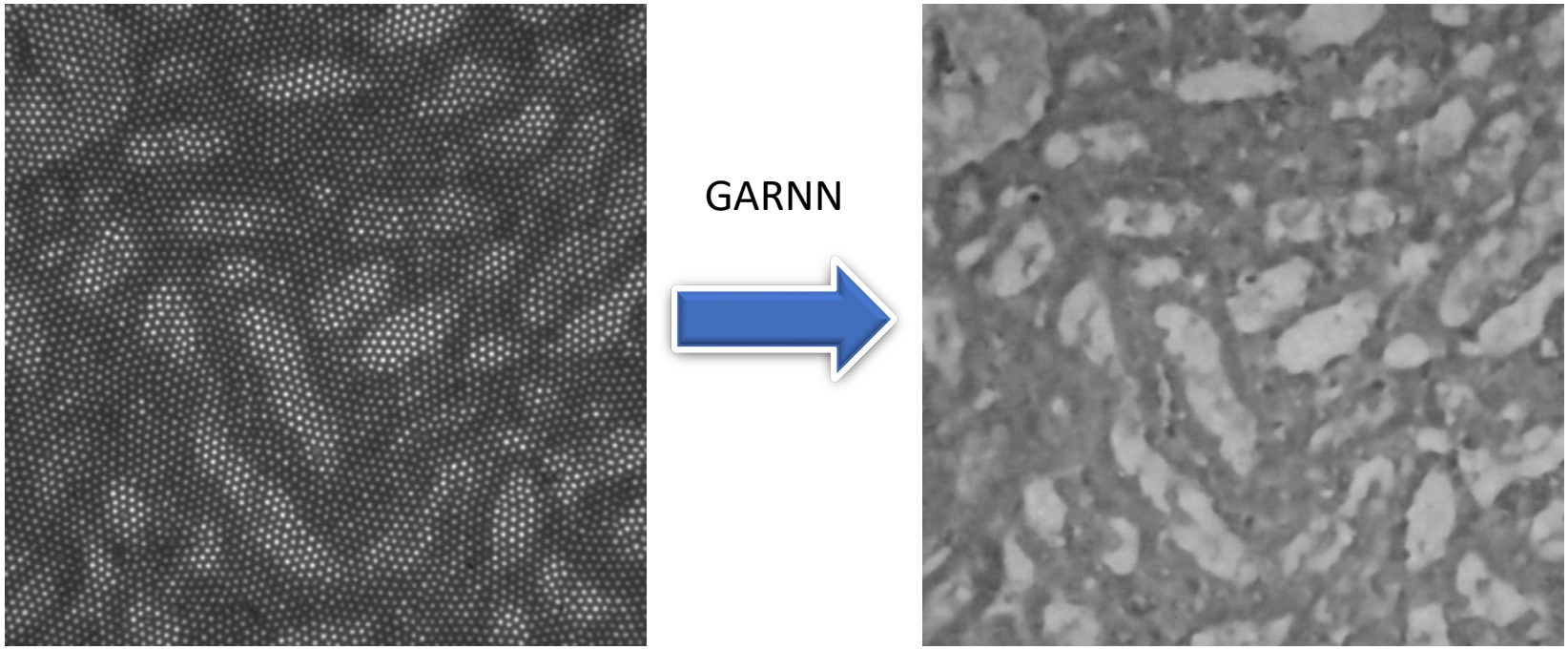
Previous proposed method

Although this method shows great resolution enhancement, it is time-consuming and multiple continuous frames may not be possible on some application scenarios!

Recent breakthroughs in deep learning have enabled us a new path of restoring high quality images from their noisy measurements.

FB restoration network

We propose a **generative adversarial restoration neural network** (**GARNN**) for FB image restoration.



Dual-sensor imaging system

Our hardware that captures FB image and its well-registered GT image.

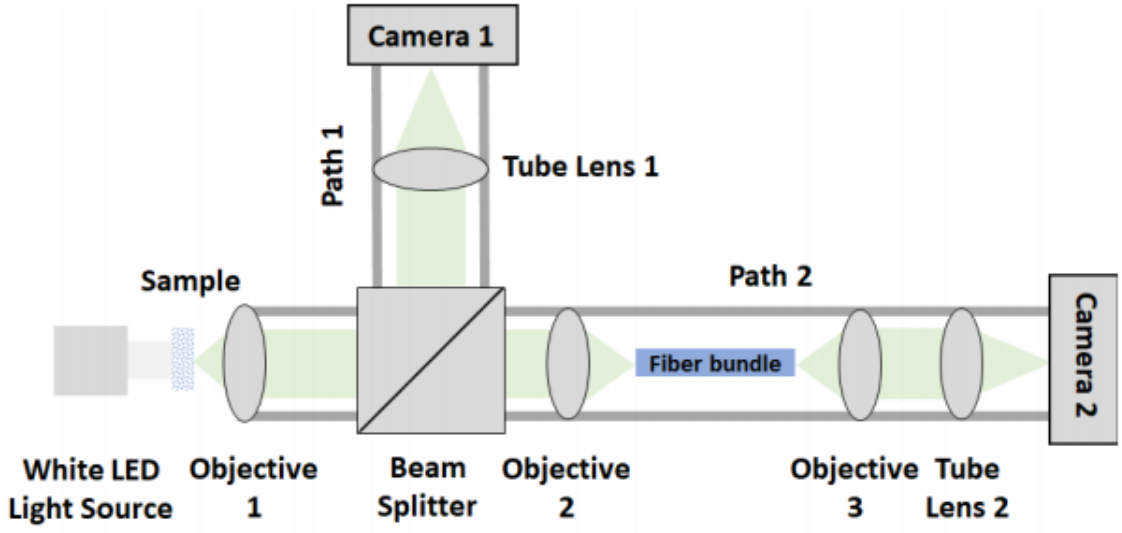
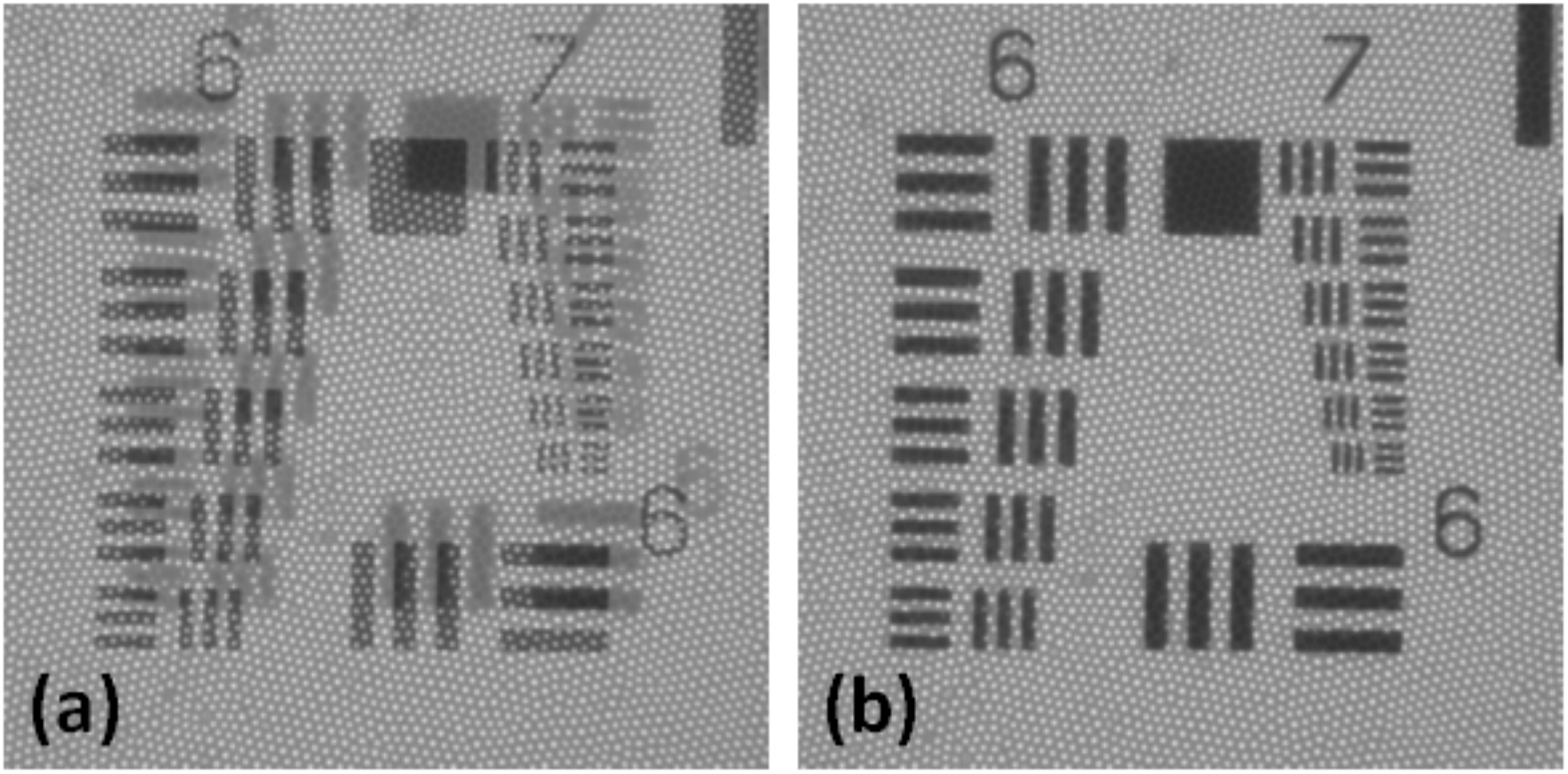


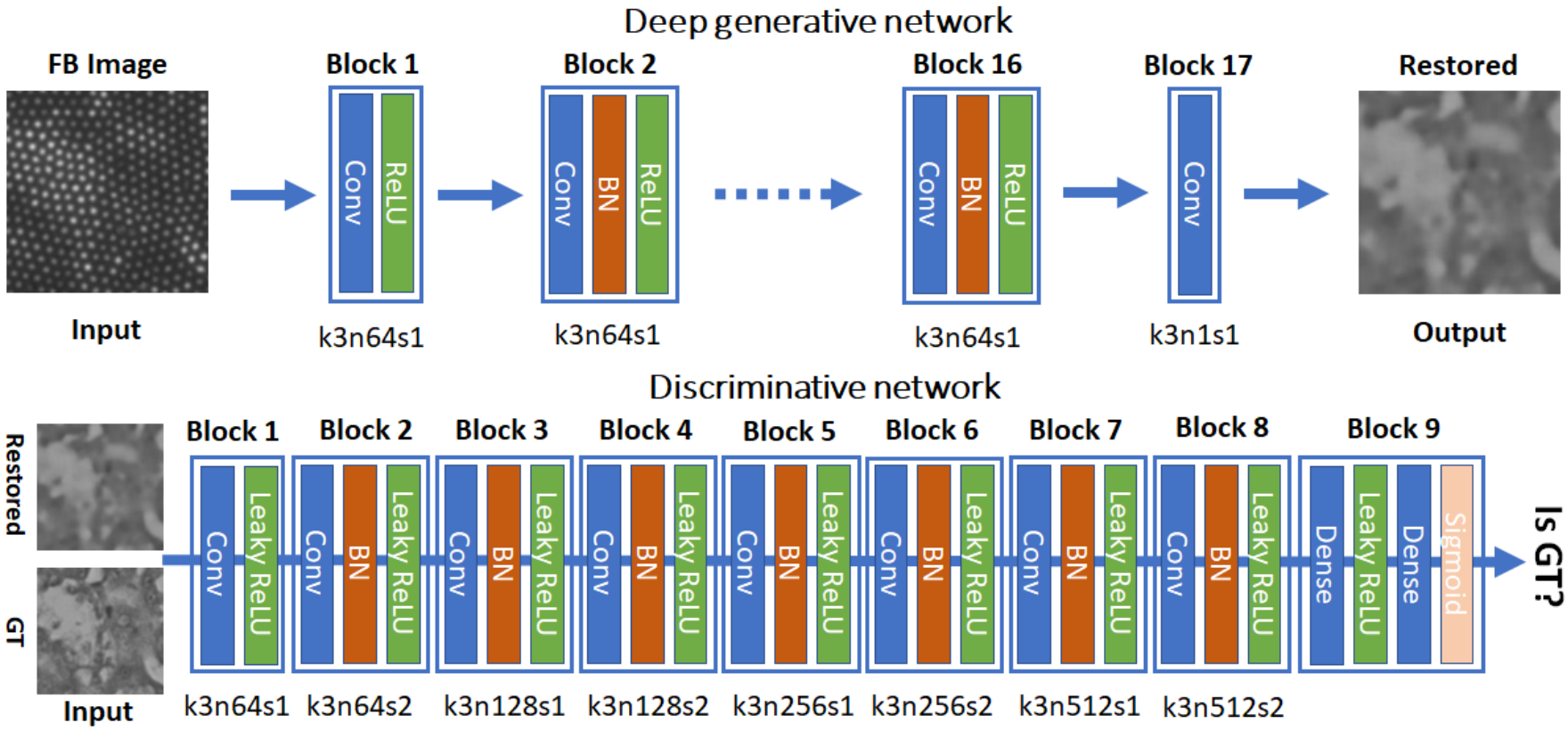
Fig. 1. Dual-sensor imaging system for capturing raw FB images and corresponding GT data for training neural network.

Dual-sensor imaging system



Overlay images of Camera 1 & 2: (a) taken from uncalibrated system; (b) taken from calibrated system.

Network Architecture



k is kernel size, n represents number of filters and s is stride size for each convolutional layer.

Network Architecture

Our proposed GARNN network consists of two networks, generative network $\mathcal{G}$ and discriminator network $\mathcal{D}$.

We create our generative network by following the design in [1]. We calculate its objective function by its content loss among FB and GT images:

$$L_{\text{Content}} = \frac{1}{N} \sum \|\mathbf{X} - \mathcal{G}(\mathbf{G})\|_2^2 ,$$

Where $\|\cdot\|_2^2$ denotes $L2$ norm, $\mathcal{G}(G)$ represents generative network's output image with FB image $\mathbf{G}$ as input. $\mathbf{X}$ is the ground truth image and N is the number of image pairs for one training iteration.

1. Zhang, K., Zuo, W., Chen, Y., Meng, D., & Zhang, L. (2017). Beyond a Gaussian denoiser: Residual learning of deep CNN for image denoising. *IEEE Transactions on Image Processing*, *26*(7), 3142–3155.

Network Architecture

Thus to obtain sharper and more realistic SR images, we add a discriminative network $\mathcal{D}$ by following the design in [1] to perform adversarial learning. The loss function of this discriminative network is:

$$L_{\mathcal{D}} = E[\mathcal{D}(\mathbf{X}), 1] + E[\mathcal{D}(\mathcal{G}(\mathbf{G})), 0],$$

$E[\mathcal{D}(X), 1]$ is the binary cross entropy between the discriminator's prediction on GT image and the desired label 1. $E[\mathcal{D}\big(\mathcal{G}(G)\big), 0]$ is the binary cross entropy between discriminator's decision on estimated image from the generative network and the desired label 0.

1. Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., … Shi, W. (2016). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network.

Network Architecture

Finally by adding the content loss and adversarial loss, we formulate the generative loss function of our GADNN as:

$$L_{\mathcal{G}} = L_{\text{Content}} + \beta E[\mathcal{D}(\mathcal{G}(\mathbf{G})), 1],$$

During training, we first initialize the network by only minimizing the content loss with updating the trainable parameters in network $\mathcal{G}$. After then, we alternately minimize $L_{\mathcal{D}}$and $L_{\mathcal{G}}$ and trainable parameters in both $\mathcal{G}$ and $\mathcal{D}$ are updated. We will only use trained network $\mathcal{G}$ for testing.

Quality measurement

As noted by others (e.g., [1]) traditional metrics PSNR or SSIM is not necessarily a good proxy for perceptual image quality.

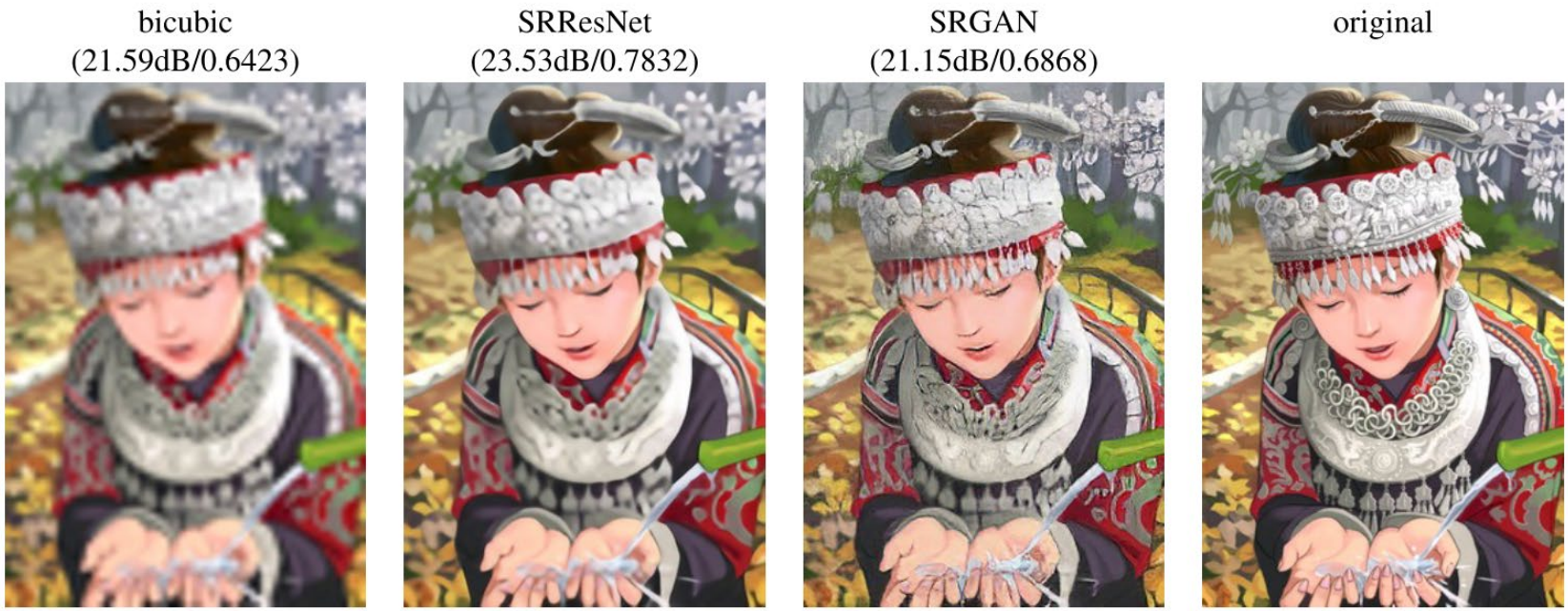


Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

1. Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., … Shi, W. (2016). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network.

Quality measurement

One recent paper [1] suggests two promising objective perceptual metrics:
Learned Perceptual Image Patch Similarity (LPIPS) metric
Berkeley Adobe Perceptual Patch Similarity (BAPPS)

In our application, we alternatively conduct subjective mean opinion score (MOS) studies. We plan to ask 10 raters to score all the SR images with an integral grade from 1 to 5 (1 for bad and 5 for best), based on sharpness and visual image fidelity compared to GT data.

1. Zhang, Richard, et al. "The unreasonable effectiveness of deep features as a perceptual metric." arXiv preprint (2018)

Normalization

Data normalization: Each image was whitened by subtract mean and divide STD. Predicted lines were used for restoring test images. This is essential for training a brightness-invariant model!

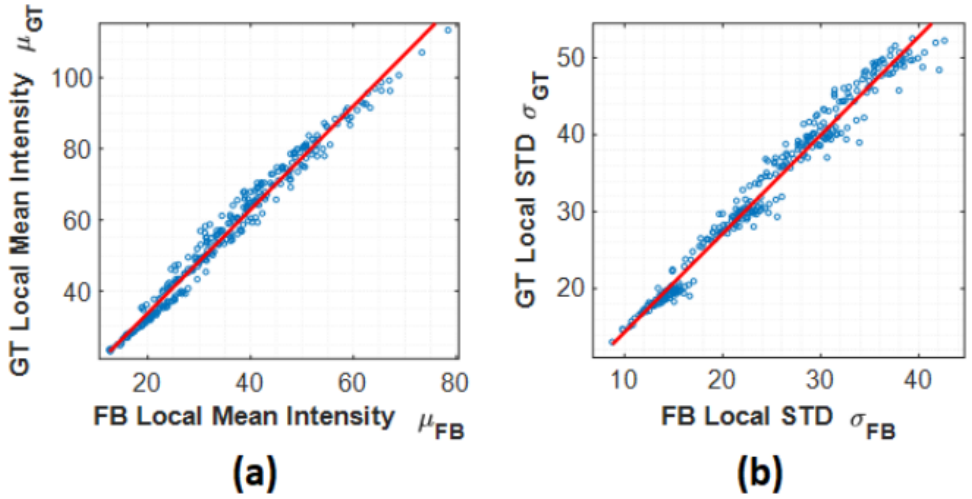


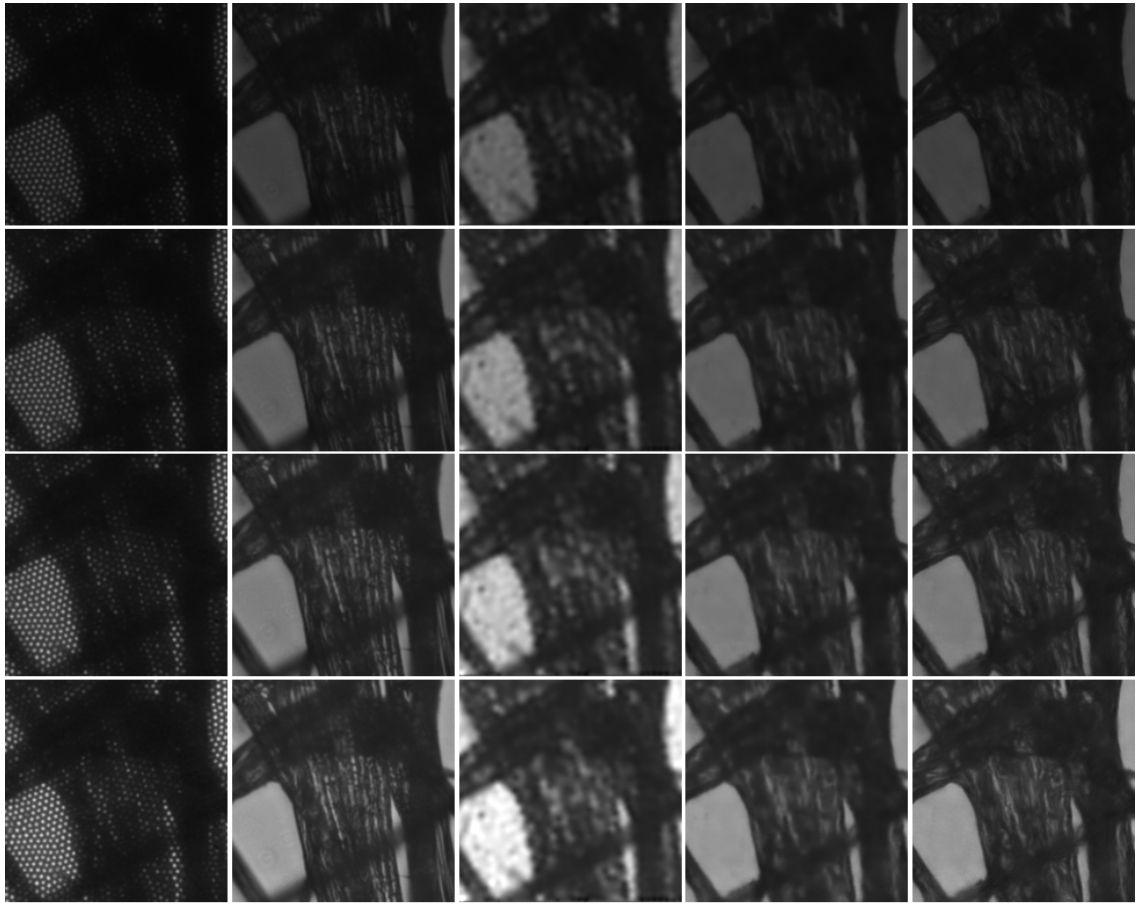
Fig. 3. Plots for μ and σ mapping relationships from FB to GT images in lens tissue training dataset: (a) mean intensity μ mapping; (b) STD σ mapping. Each blue dot is from one captured image. Red lines mark the predicted linear models of μ and σ respectively.

Dataset: lens tissue

Lens tissue sample

79 pairs of 800*800 images for training
19 pairs of 800*800 images for testing.

Dataset: lens tissue



(a) (b) (c) (d) (e)

Fig. 4. Experimental results with lens tissue under four illuminations conditions: (a) raw FB images, (b) GT images, (c) results from MAP, (d) results from RED30, (e) results from GRNN , and (f) results from GARNN. Brightness increases from top to bottom.

Dataset: lens tissue

Table 1. Average PSNR, SSIM & MOS results for lens tissue test dataset.

	FB	MAP	RED30	GRNN	GARNN
PSNR	19.4 dB	17.5 dB	**31.8** dB	31.7 dB	31.4 dB
SSIM	0.54	0.80	**0.89**	**0.89**	0.87
MOS	N/A	1.35	3.13	3.13	**3.64**

Table 2. Cross-validation experiments for lens tissue sample.

	Level 1	Level 2	Level 3	Level 4	Mean
TAF	35.1 dB	32.4 dB	30.1 dB	28.1 dB	31.4 dB
LOOF	35.1 dB	32.1 dB	30.0 dB	27.7 dB	31.2 dB

Dataset: human slides

Human histological sample:

146 pairs of 800*800 images for training
24 pairs of 800*800 images for testing.

Dataset: human slides

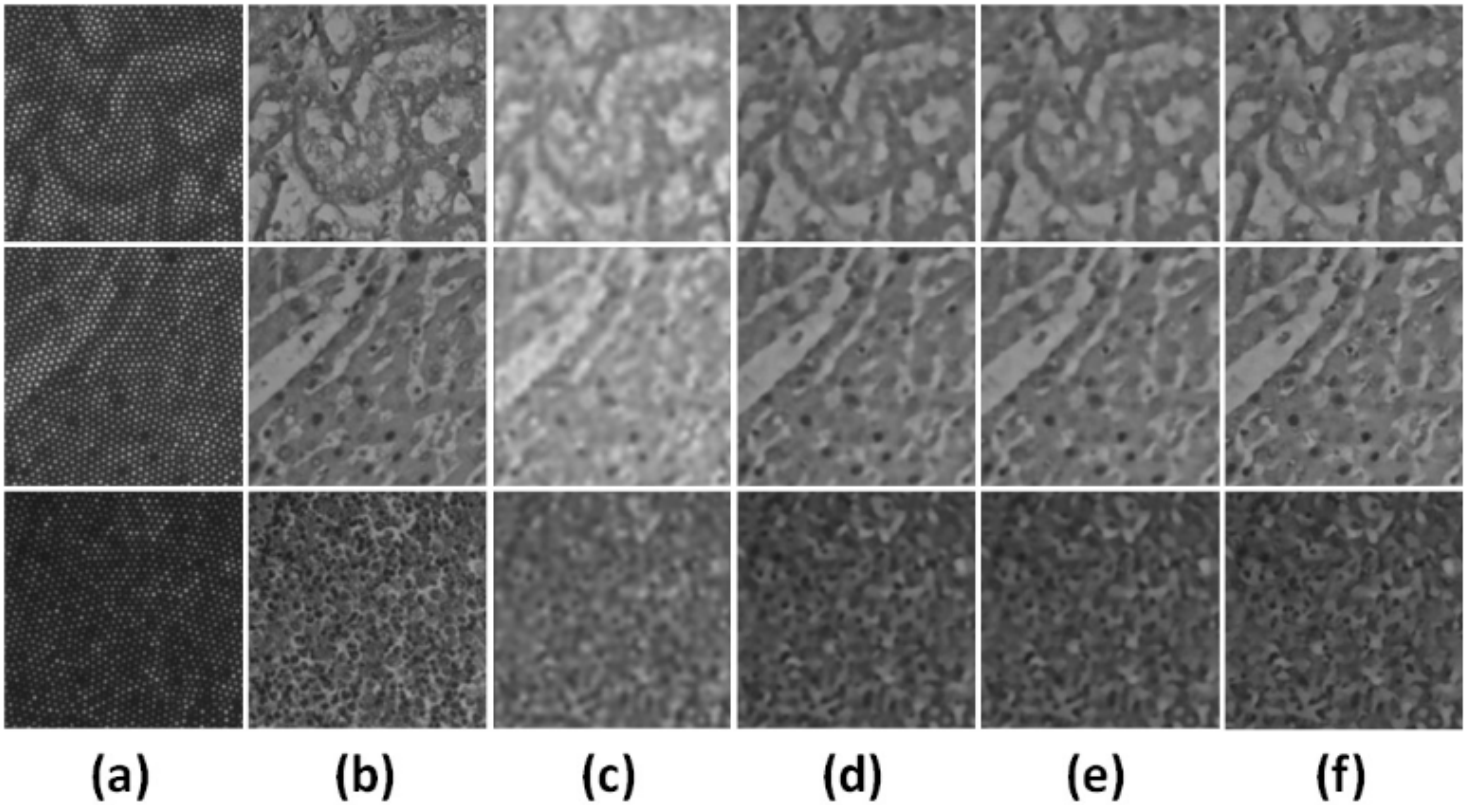


Fig. 6. Experimental results with human histological specimens (from top to bottom: kidney, liver, and tonsil): (a) raw FB images, (b) GT images, (c) results from MAP, (d) results from RED30, (e) results from GRNN, and (f) results from GARNN.

Dataset: human slides

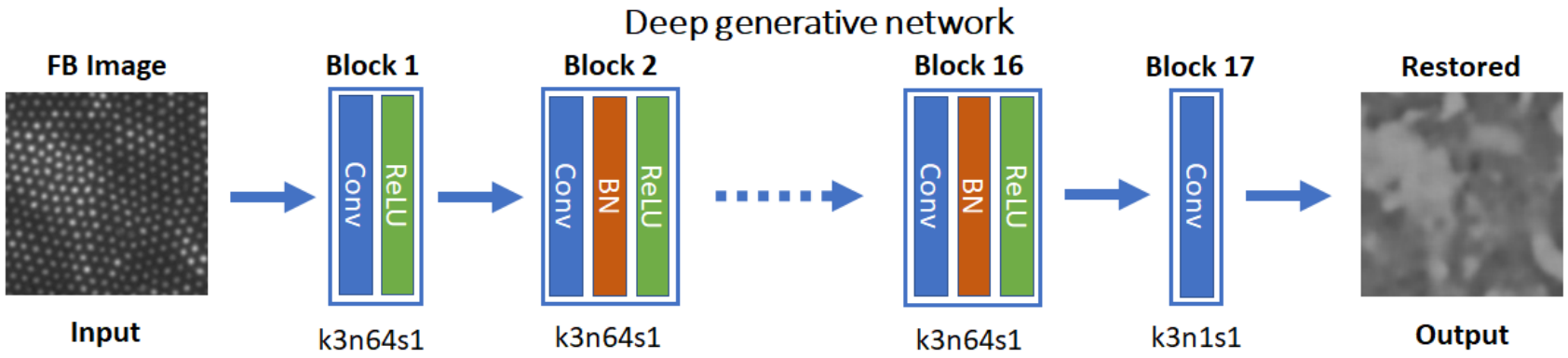
Table 3. PSNR, SSIM & MOS results for human specimens test dataset.

		FB	MAP	RED30	GRNN	GARNN
Kidney	PSNR	15.9 dB	18.7 dB	**30.1** dB	30.0 dB	29.8 dB
	SSIM	0.33	0.77	**0.83**	**0.83**	0.83
	MOS	N/A	1.21	2.70	2.70	**3.50**
Liver	PSNR	16.6 dB	18.6 dB	**30.3** dB	**30.3** dB	30.2 dB
	SSIM	0.34	0.77	**0.84**	**0.84**	0.83
	MOS	N/A	1.13	2.51	2.51	**3.60**
Tonsil	PSNR	17.1 dB	24.6 dB	**25.7** dB dB	**25.7** dB	25.5 dB
	SSIM	0.42	0.69	**0.72**	**0.72**	**0.72**
	MOS	N/A	1.32	2.21	2.21	**2.52**

Implementation details

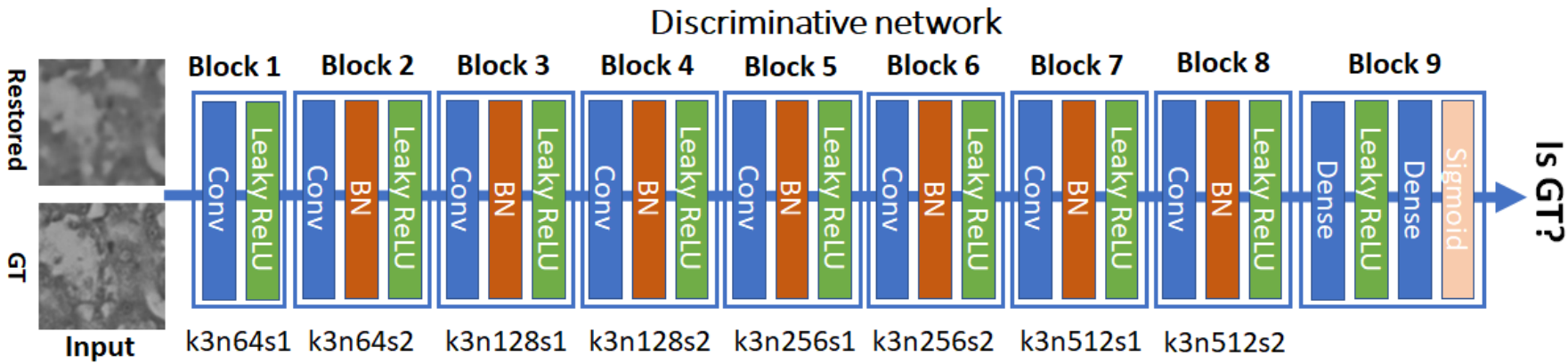
Environment: TensorFlow container on HPC.
Hardware: Nvidia P100 GPU.
Automatic data acquisition: LabView
Real-time overlay stream: OpenCV

Implementation details




```
def grnn(input, is_training=True, feature_size = 64, layer_depth = 17, output_channels=1):
    with tf.variable_scope('block1'):
        output = tf.layers.conv2d(input, feature_size, 3, padding='same', activation=tf.nn.relu)
    for layers in range(2, layer_depth):
        with tf.variable_scope('block%d' % layers):
            output = tf.layers.conv2d(output, feature_size, 3, padding='same', name='conv%d' % layers, use_bias=False)
            output = tf.nn.relu(tf.layers.batch_normalization(output, training=is_training))
    with tf.variable_scope('block%d' % layer_depth):
        output = tf.layers.conv2d(output, output_channels, 3, padding='same')
    return output
```

Implementation details




```
def Discrim_Net(input,is_train=False,reuse=False):
        w_init = tf.random_normal_initializer(stddev=0.02)
        with tf.variable_scope('Discriminator_Layer',reuse=reuse):
            output = tf.layers.conv2d(input,64,3,(1,1),padding='SAME',activation=tf.nn.leaky_relu,use_bias=False,
                                      kernel_initializer=w_init)

            output = tf.layers.conv2d(output,64,3,(2,2),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))

            output = tf.layers.conv2d(output,128,3,(1,1),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))
            output = tf.layers.conv2d(output,128,3,(2,2),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))

            output = tf.layers.conv2d(output,256,3,(1,1),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))
            output = tf.layers.conv2d(output,256,3,(2,2),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))

            output = tf.layers.conv2d(output,512,3,(1,1),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))
            output = tf.layers.conv2d(output,512,3,(2,2),padding='SAME',use_bias=False,kernel_initializer=w_init)
            output = tf.nn.leaky_relu(tf.layers.batch_normalization(output, training=is_train))

            output = tf.layers.flatten(output)
            output = tf.layers.dense(output,1024,activation=tf.nn.leaky_relu)
            output = tf.layers.dense(output,1)
            return output
```

Implementation details

Batch Normalization [1], adds two trainable parameters: gamma and beta. It can accelerate convergence.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

1. Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." ***arXiv preprint arXiv:1502.03167*** **(2015).**

Implementation details

During training, we first initialize the network by only minimizing the content loss with updating the trainable parameters in network $\mathcal{G}$. After then, we alternately minimize $L_{\mathcal{D}}$ and $L_{\mathcal{G}}$ and trainable parameters in both $\mathcal{G}$ and $\mathcal{D}$ are updated. We will only use trained network $\mathcal{G}$ for testing.

```
# loss function
with tf.name_scope('DNet_loss'):
    d_loss1 = tf.nn.sigmoid_cross_entropy_with_logits(logits=bool_T, labels=tf.ones_like(bool_T))
    d_loss2 = tf.nn.sigmoid_cross_entropy_with_logits(logits=bool_F, labels=tf.zeros_like(bool_F))
    self.d_loss = tf.reduce_mean(d_loss1) + tf.reduce_mean(d_loss2)
with tf.name_scope('loss'):
    loss_d = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=bool_F, labels=tf.ones_like(bool_F)))
    loss_g = (1.0 / batch_size) * tf.nn.l2_loss(tf.subtract(self.Y, self.Y_))
    self.loss = loss_g  + Discrim_Rate * loss_d
with tf.name_scope('loss_init'):
    self.loss_init = loss_g
```

```
# Optimizer
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
    self.train_op_init = tf.train.AdamOptimizer(self.lr_g).minimize(self.loss_init, global_step=self.global_step_g,
                                                                    var_list=var_list_g)
    self.train_op_d = tf.train.AdamOptimizer(self.lr_d).minimize(self.d_loss, global_step=self.global_step_d,
                                                                 var_list=var_list_d)
    self.train_op = tf.train.AdamOptimizer(self.lr_d).minimize(self.loss, global_step=self.global_step_d,
                                                               var_list=var_list_g)
```

Thank You