Image Super-Resolution Using Deep Convolutional Networks

Dept. of Electronic Engineering
Sogang University

Song-Woo Choi

Introduction

- Single image super-resolution
  - It produces high-resolution image from a single low-resolution image
  - Ill-posed problem(underdetermined inverse problem)
    - Effectively fewer equations than unknown; solution is not unique

540*120

Single LR image

Underdetermined

SR

1080*240

Single HR image

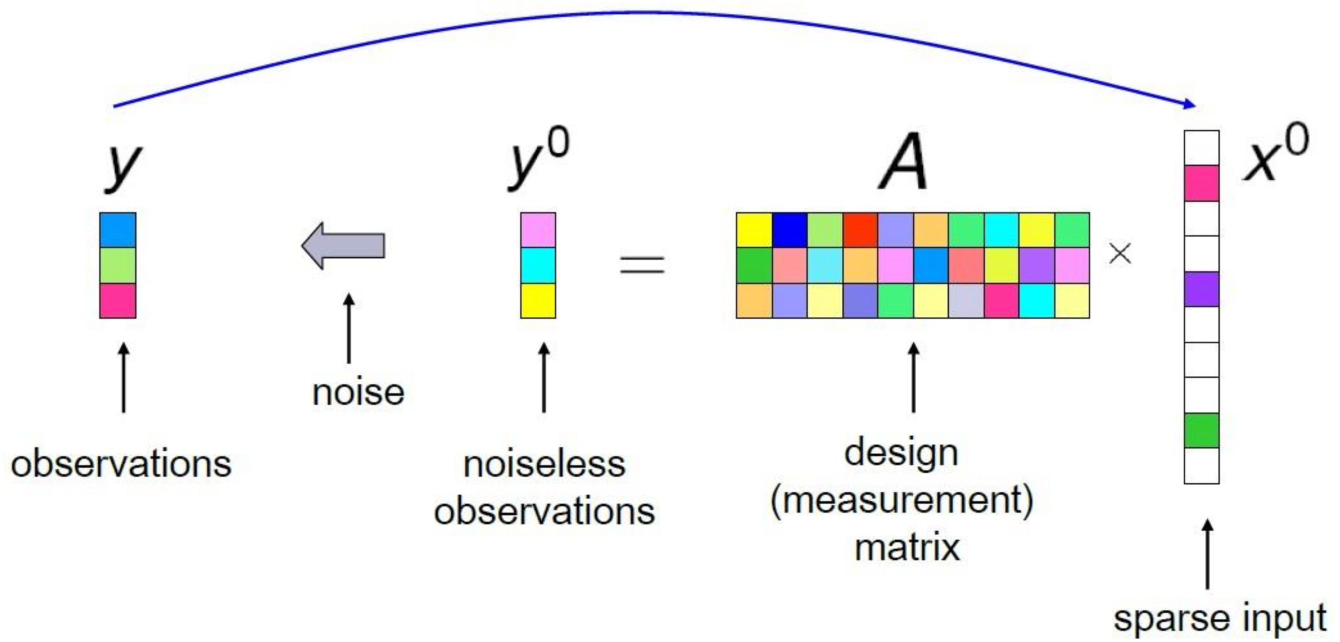
Let A be an $m \times n$ matrix, $b \in \mathbb{R}^m$ be a vector.
A linear $system$: $Ax = b$.

$$\begin{aligned} x+3y+2z&=2 \\ x+y+z&=4 \end{aligned} \quad \Longrightarrow \quad \begin{bmatrix} 1 & 3 & 2 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Since $m < n$,
This system is underdetermined

Sparse Representation

- Sparse dictionary learning
  - A representation learning method which aims at finding a sparse representation of the input data in the form of a linear combination of basic elements

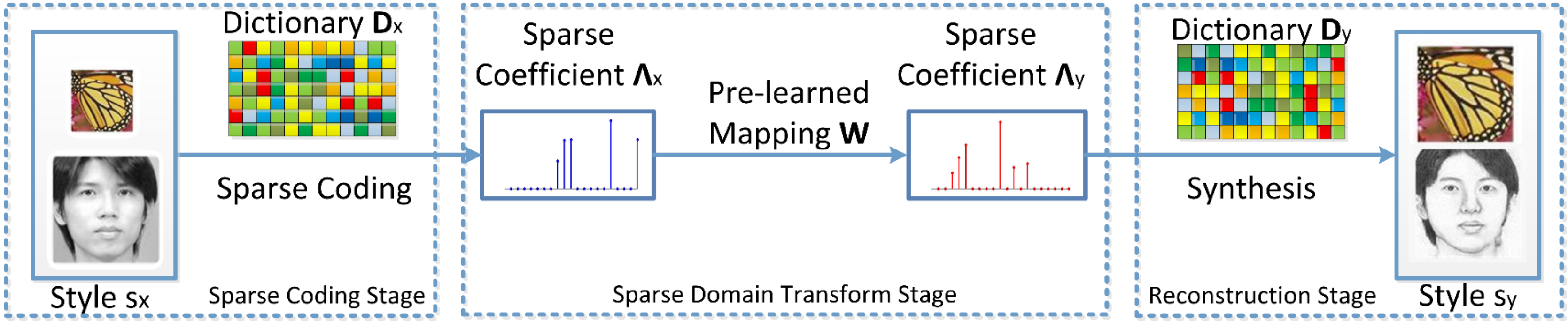


Conventional method

- Single image super-resolution(example based)[1]
  - Extract overlapping patches (overlapped tiling) (y_l)
  - For each patch find the low resolution representation using D_l

    $$x^* = \underset{x}{\operatorname{argmin}} \frac{1}{2}||y_l - D_l x||_2^2 + \lambda||x||_1$$

    Find the sparse linear representation of low resolution patch based on LR dictionary
  - Find the high resolution representation using D_h and the same x^*
  - Construct the high resolution image from high-res patches



Proposed Method

- Deep learning(CNNs) based SR[2]
  - A pipeline of dictionary learning based SR[1] is equivalent to a deep convolutional neural network
    - A CNN directly **learns** an **end-to-end mapping** between **low-** and **high-**resolution images
  - Why SR-CNN?
    - Simple structure, superior accuracy
    - Fast speed for practical on-line usage even on a CPU
      - Fully feed-forward network and not solving any optimization problem
    - Restoration quality can be further improved when,
      1. Larger and more diverse datasets are available
      2. A larger and deeper model is used

Proposed Method

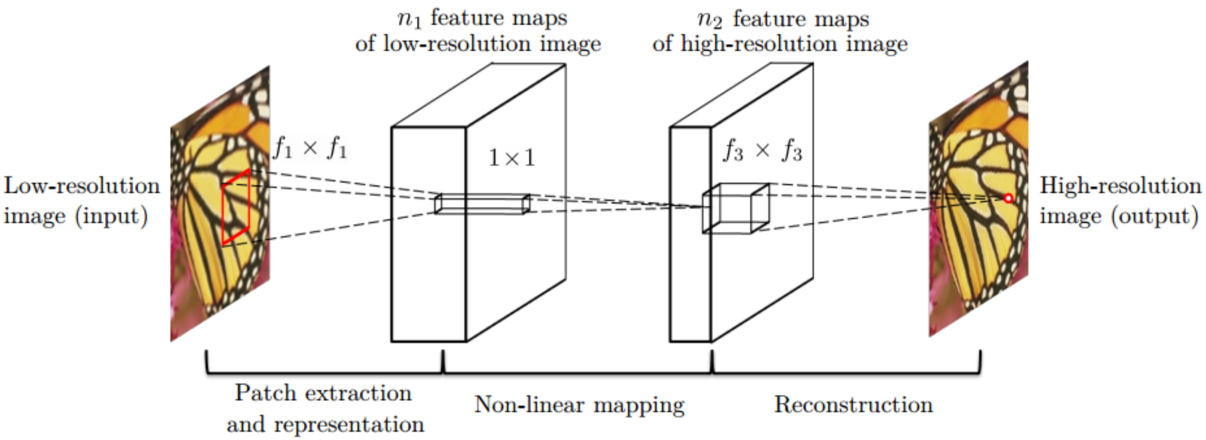
- CNN for SR
  - Formulation
    1. Patch extraction and representation

       $F_1(\mathbf{Y}) = \max(0, W_1 * \mathbf{Y} + B_1)$; W_1: filters($c \times f_1 \times f_1 \times n_1$), B_1: biases
    2. Non-linear mapping

       $F_2(\mathbf{Y}) = \max(0, W_2 * F_1(\mathbf{Y}) + B_2)$; W_2: filters($n_1 \times 1 \times 1 \times n_2$)
    3. Reconstruction

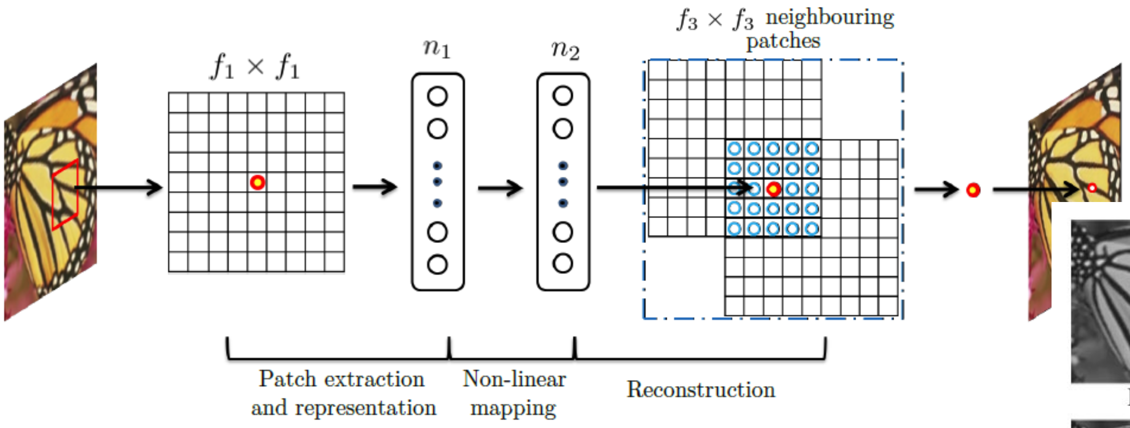
       $F(\mathbf{Y}) = W_3 * F_2(\mathbf{Y}) + B_3$; W_3: filters($n_2 \times f_3 \times f_3 \times c$)

ReLU: $f(x) = \max(0, x)$



Proposed Method

- Relation to sparse-coding
  - Patches with size of $f_1 \times f_1$ extracted from image
  - Find a n_2 sparse set of coefficients in a n_1 sized dictionary
  - Reconstruct a high resolution patch from corresponding HR patches with found weight



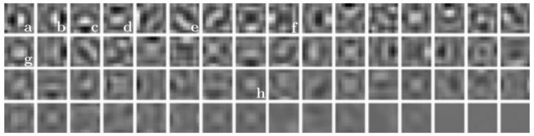
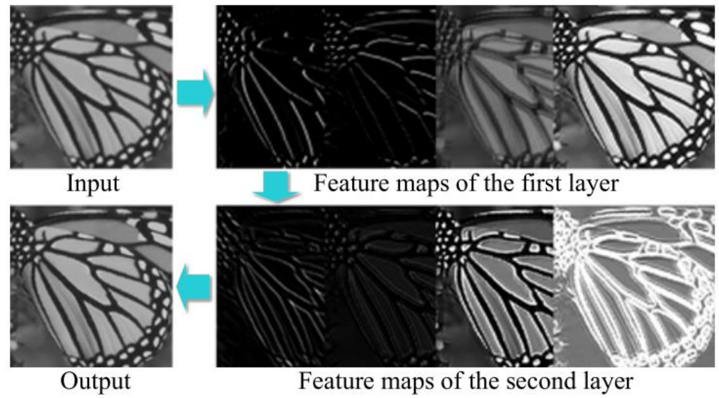


Fig. 5. The figure shows the first-layer filters trained on ImageNet with an upscaling factor 3. The filters are organized based on their respective variances.



Proposed Method

- Training
  - Estimation of network parameters
    - $\Theta = \{W_1, W_2, W_3, B_1, B_2, B_3\}$
    - Loss function (MSE): favoring a high PSNR

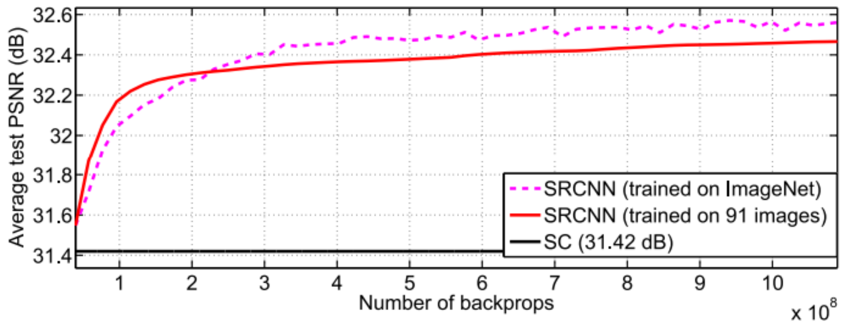
sub-image cropped from the training image

$$L(\Theta) = \frac{1}{n}\sum_{i=1}^{n} ||F(\mathbf{Y}_i; \Theta) - \mathbf{X}_i||^2$$

Up-scaled image after blurring by a Gaussian kernel

    - Optimization technique for backpropagation (SGD)

$$\Delta_{i+1} = 0.9 \cdot \Delta_i + \eta \cdot \frac{\partial L}{\partial W_i^l}, W_{i+1}^l = W_i^l + \Delta_{i+1}$$



Experimental Results

- Code(test)

```
def run_benchmark():

  with tf.Session() as sess:
    images = tf.placeholder(tf.float32, shape=(None, FLAGS.image_size, FLAGS.image_size, FLAGS.num_channels))
    labels = tf.placeholder(tf.float32, shape=(None, FLAGS.label_size, FLAGS.label_size, FLAGS.num_channels))

    outputs, weight_parameters, bias_parameters = inference(images)

    global_step = tf.Variable(0)

    loss = tf.sqrt(tf.reduce_mean(tf.square(tf.sub(labels, outputs))))
    # print (outputs.get_shape())
    # train_op1 = tf.train.GradientDescentOptimizer(0.0001).minimize(loss, global_step = global_step)
    op1 = tf.train.GradientDescentOptimizer(0.0001)
    op2 = tf.train.GradientDescentOptimizer(0.0001*0.1)
    grads = tf.gradients(loss, weight_parameters + bias_parameters)
    grads1 = grads[:len(weight_parameters)]
    grads2 = grads[len(weight_parameters):]
    train_op1 = op1.apply_gradients(zip(grads1, weight_parameters), global_step = global_step)
    train_op2 = op2.apply_gradients(zip(grads2, bias_parameters), global_step = global_step)
    train_op = tf.group(train_op1, train_op2)

    saver = tf.train.Saver(weight_parameters + bias_parameters)

    init = tf.initialize_all_variables().run()

    train_data, train_label = read_data('train.h5')
    train_data = np.transpose(train_data, (0,2,3,1))
    train_label = np.transpose(train_label, (0,2,3,1))
    data_size = int(train_data.shape[0] / FLAGS.batch_size)
    # print (train_data.shape)
    num_steps_burn_in = 10
    step = 0
    for i in xrange(FLAGS.num_iter):
      start_time = time.time()
      batch_data = train_data[(i % data_size) * FLAGS.batch_size : ((i+1) % data_size) * FLAGS.batch_size, :,:,:]
      batch_label = train_label[(i % data_size) * FLAGS.batch_size : ((i+1) % data_size) * FLAGS.batch_size, :,:,:]
      # print (batch_label.shape)
      _,step = sess.run([train_op, global_step], feed_dict={images:batch_data, labels:batch_label})
      duration = time.time() - start_time
      # if i > num_steps_burn_in:
      if not i % num_steps_burn_in:
        saver.save(sess, 'my-model', global_step=i)
        print ('%s: step %d, duration = %.3f' %
               (datetime.now(), i, duration))
```

Experimental Results

- Code(test)

```
function im_h = SRCNN(model, im_b)

%% load CNN model parameters
load(model);
[conv1_patchsize2,conv1_filters] =
size(weights_conv1);
conv1_patchsize = sqrt(conv1_patchsize2);
[conv2_channels,conv2_patchsize2,conv2_filters] =
size(weights_conv2);
conv2_patchsize = sqrt(conv2_patchsize2);
[conv3_channels,conv3_patchsize2] =
size(weights_conv3);
conv3_patchsize = sqrt(conv3_patchsize2);
[hei, wid] = size(im_b);

%% conv1
weights_conv1 = reshape(weights_conv1,
conv1_patchsize, conv1_patchsize, conv1_filters);
conv1_data = zeros(hei, wid, conv1_filters);
for i = 1 : conv1_filters
    conv1_data(:,:,i) = imfilter(im_b,
weights_conv1(:,:,i), 'same', 'replicate');
    conv1_data(:,:,i) = max(conv1_data(:,:,i) +
biases_conv1(i), 0);
end
```

```
%% conv2
conv2_data = zeros(hei, wid, conv2_filters);
for i = 1 : conv2_filters
    for j = 1 : conv2_channels
        conv2_subfilter =
reshape(weights_conv2(j,:,i), conv2_patchsize,
conv2_patchsize);
        conv2_data(:,:,i) = conv2_data(:,:,i) +
imfilter(conv1_data(:,:,j), conv2_subfilter, 'same',
'replicate');
    end
    conv2_data(:,:,i) = max(conv2_data(:,:,i) +
biases_conv2(i), 0);
end

%% conv3
conv3_data = zeros(hei, wid);
for i = 1 : conv3_channels
    conv3_subfilter = reshape(weights_conv3(i,:),
conv3_patchsize, conv3_patchsize);
    conv3_data(:,:) = conv3_data(:,:) +
imfilter(conv2_data(:,:,i), conv3_subfilter, 'same',
'replicate');
end

%% SRCNN reconstruction
im_h = conv3_data(:,:) + biases_conv3;
```

Experimental Results

- Result images

SR-CNN

Original

Experimental Results

- Result images

Experimental Results

- Result images (Original)

Experimental Results

- Result images (Bicubic interpolation: 2x upscaling)

Experimental Results

- Result images (SRCNN : 2x upscaling)