Korean Advanced Institute of Science and Technology

School of Electrical Engineering

# EE838A

## Special Topics on Image Engineering:

## Advanced Image Restoration and Quality Enhancement

## Fall 2018

Student's Name: Dinh Vu

Student's ID: 20184187

Student's Cellphone: 010-2975-8980

Student's Email: dinhvu@kaist.ac.kr

# Homework 1

## Implementation and Verification of Single Image Super-Resolution (SISR)

### 1. The architecture of the model

In my source code, all functions are written by Python and based on Tensorflow framework. The required structure for SISR is depicted in the following Figure 1.1.

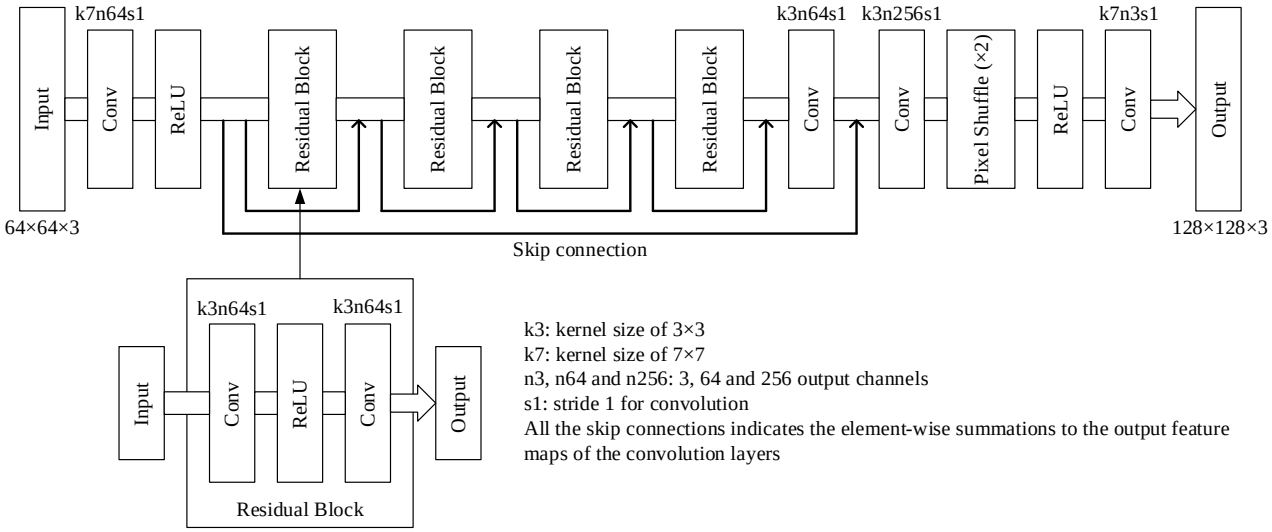


**Figure 1.1.** The SISR structure

The convolution layers are implemented by a simple function `tf.layers.conv2d()`, such as the first convolution layer k7n64s1 below.

```
x = tf.layers.conv2d(x, 64, 7, strides=(1,1), padding="same", name="k7n64s1")
```

In order to preserve the resolution of the feature maps 64×64 before the Pixel Shuffle layer, the convolution has to be operated with zero padding, then the parameter `padding` is set to "`same`". The first ReLU layer is built by the function `tf.nn.relu()`

```
x = tf.nn.relu(x, name="relu_1")
```

Because there are 4 Residual Blocks with 4 corresponding skip connections, the for-loop is used to avoid repeating. The skip connections, indicating that the element-wise summations, can be presented by `tf.add()` function.

```
for i in range(4):
   xx = tf.layers.conv2d(x, 64, 3, strides=(1,1), padding="same",
name="res_%s/k3n64s1_in" % i)
   xx = tf.nn.relu(xx, name="res_%s/relu" % i)
   xx = tf.layers.conv2d(xx, 64, 3, strides=(1,1), padding="same",
name="res_%s/k3n64s1_out" % i)
   xx = tf.add(x, xx)
   x = xx
```

After the convolution layer k3n256s1, for the Pixel Shuffle (×2) layer, the input feature map has size of 64×64×256 and the shape of the output feature map is 128×128×64. In Tensorflow, `tf.depth_to_space()` is the function reshaping the features from the dimension of depth to the spatial dimensions.

```
x = tf.depth_to_space(x, scale) # Pixel shuffle layer (x2), scale=2
```

After the following ReLU and convolution layer, the final output shape of the model is 128×128×3. The `padding` parameter is still set to "`same`" to reserve the resolution of feature map 128×128.

```
x = tf.nn.relu(x, name="relu_2")
x = tf.layers.conv2d(x, 3, 7, strides=(1,1), padding="same", name="k7n3s1")
```

The structure of model is presented completely in `sisr_model.py` file.

## 2. Training Process

### *2.1. Data pipeline*

Two placeholders are created for feed data, one for LR patches and other for HR patches.

```
x = tf.placeholder(tf.float32, [None,patch_size,patch_size,3], name='lr_input')
y = tf.placeholder(tf.float32, [None,scale*patch_size,scale*patch_size,3],
name='hr_target')
```

Before training, the input pipeline must be built in order to feed data into the model. Therefore, the Tensorflow's Dataset module `tf.data` is chosen to build efficient pipelines for images. First, the list of directories of the training images is got by `get_filepath()` function.

From line 31 of `utils.py` file:

```
def get_filepath(path, suffix):
   file_path = []
   for f in os.listdir(path):
      if f.endswith(suffix):
         file_path.append(os.path.join(path, f))
   file_path = sorted(file_path)
   return file_path
```

From line 35 of `train.py` file:

```
train_hr_path = get_filepath(train_hr_dir, '.png')
```

```
n_trains = len(train_hr_path)
train_hr_path = tf.constant(train_hr_path)
```

In the training process, only the High Resolution (HR) images are used and each Low Resolution (LR) image can be interpolated from the corresponding HR image by Bicubic algorithm because the given training LR images are also generated from `imresize()` function in Matlab. After getting the filename of the training HR images, the following steps are applied:

- Create dataset from slices of `train_hr_path`
- Shuffle the data with a buffer size equals to the length of the dataset
- Parse the images from the list of directories to the pixel values
- Batch the images

```
dataset = tf.data.Dataset.from_tensor_slices(train_hr_path)
dataset = dataset.shuffle(n_trains)   # n_trains = 4900
dataset = dataset.map(train_parse)
dataset = dataset.batch(batch_size)   # batch_size = 16
```

The `train_parse()` function will do the following:

- Read the contents of training image
- Decode using png format
- Random crop 128×128 patch from the HR image
- Generate 64×64 LR patch from HR patch by Bicubic algorithm in Matlab
- Convert data type of pixel from uint8 to float32
- Rescale pixel values in LR patch and HR patch from range [0, 255] to range [0, 1]

```
def train_parse(hr_dir):
  hr_string = tf.read_file(hr_dir)
  hr_decoded = tf.image.decode_png(hr_string, channels=3)
  hr_patch = tf.random_crop(hr_decoded, [scale*patch_size, scale*patch_size, 3])

  lr_patch = tf.py_func(resize_img, [hr_patch], tf.uint8)
  lr_patch = tf.reshape(lr_patch, [patch_size,patch_size,3])

  lr_patch = tf.image.convert_image_dtype(lr_patch, tf.float32)
  hr_patch = tf.image.convert_image_dtype(hr_patch, tf.float32)

  return lr_patch, hr_patch
```

To feed patches to the graph, the iterator will be created and the `get_next()` operation will call the next batch

```
iter = dataset.make_initializable_iterator()
lr_patch, hr_patch = iter.get_next()
```

### *2.2. Optimization*

The L1 loss is defined below and to visualize the loss value in tensorboard, the function `tf.summary.scalar()` is called.

```
with tf.name_scope('L1_Loss'):
   loss = tf.losses.absolute_difference(pred, y)
```

```
tf.summary.scalar("loss", loss)
```

Using Adam Optimization is a requirement, however the parameters of Adam Optimizer can be chosen and the chosen values are $\beta_1 = 0.7; \beta_2 = 0.3; \varepsilon = 10^{-8}$.

```
optimizer = tf.train.AdamOptimizer(lr_v, beta1, beta2)
train_op = optimizer.minimize(loss)
```

The learning rate is initialized at $10^{-4}$ and after 100 epochs the learning rate will be decrease by half.

```
if epoch != 0 and (epoch % decay_period == 0):
   new_lr_decay = lr_v * learning_rate_decay
   sess.run(tf.assign(lr_v, new_lr_decay))
   log = "** New learning rate: %.9f **\n" % (lr_v.eval())
   write_logs(logs_train, log, False)
elif epoch == 0:
   sess.run(tf.assign(lr_v, learning_rate_init))
   log = "** Initial learning rate: %.9f **\n" % (learning_rate_init)
   write_logs(logs_train, log, False)
```

The number of training images is 4900 and the batch size equals to 16, so the number of batches at each epoch is:

$$\text{n\_batches} = \left\lceil \frac{\text{n\_trains}}{\text{batch\_size}} \right\rceil = \left\lceil \frac{4900}{16} \right\rceil = 307$$

The training process will be finished after 1000 epochs. In the session, the training process is organized usually:

- Initialize all variables
- Read LR and HR patches
- Feed LR and HR patches to the graph through `tf.placeholder()`
- Calculate L1 loss and update the weights of the neural networks

```
with tf.Session() as sess:
   # Initialize variables
   sess.run(tf.global_variables_initializer())
   ...
   train_start = time.time()
   for epoch in range(n_epoch):
      epoch_start = time.time()
      ……
      avg_loss = 0
      sess.run(iter.initializer)
      for batch in range(n_batches):
         batch_start = time.time()
         lr_patches, hr_patches = sess.run([lr_patch, hr_patch])
         _, loss_val, summary = sess.run([train_op, loss, merged_summary_op],
feed_dict={x:lr_patches, y:hr_patches})
```

Then model will be saved after training by calling the `tf.train.Saver()`.

```
saver = tf.train.Saver()
```

```
with tf.Session() as sess:
   ...
   log = "\n========== Training End ==========\n"
   write_logs(logs_train, log, False)
   # Save model
   save_path = saver.save(sess, model_dir)
   log = "Model is saved in file: %s" % save_path
   write_logs(logs_train, log, False)
```

## 3. Validation

### *3.1. Data pipeline*

Two placeholders are created to feed the data. However, unlike two placeholders in the training process, the dimension of batch must equal to 1 and the height and width must be set to `None` because the resolution of the validation images is various.

```
X = tf.placeholder(tf.float32, [1,None,None,3])
Y = tf.placeholder(tf.float32, [1,None,None,3])
```

The pipeline of the validation data is similar to the training data such as getting filenames. The main differences are that both validation LR and HR images are used, the batch size equals to 1, especially, entire validation LR image will be put into the model instead of 64×64 patches. The following steps are depicted the pipeline of the validation data.

- ❖ Get the lists of filenames of LR and HR images
- ❖ Create dataset from slices of the lists
- ❖ Parse the images from filenames to the pixel values
- ❖ Batch the images
- ❖ Create one-shot iterator because the validation data is only feed one times
- ❖ Call the operation `get_next()` to read the next images

```
valid_lr_path = get_filepath(lr_dir, '.png')
valid_hr_path = get_filepath(hr_dir, '.png')
n_valid = len(valid_lr_path)

valid_lr_path = tf.constant(valid_lr_path)
valid_hr_path = tf.constant(valid_hr_path)

dataset = tf.data.Dataset.from_tensor_slices((valid_lr_path, valid_hr_path))
dataset = dataset.map(valid_parse)
dataset = dataset.batch(1)
iter = dataset.make_one_shot_iterator()
lr_img, hr_img = iter.get_next()
```

The valid_parse() function will do the following:

- ❖ Read the contents of LR and HR image
- ❖ Decode using png format
- ❖ Convert data type of pixel value to from uint8 to float32
- ❖ Rescale the pixel value from range [0, 255] to range [0, 1]

```
def valid_parse(lr, hr):
```

```
  lr_string = tf.read_file(lr)
  lr_decoded = tf.image.decode_png(lr_string, channels=3)
  lr_img = tf.image.convert_image_dtype(lr_decoded, tf.float32)

  hr_string = tf.read_file(hr)
  hr_decoded = tf.image.decode_png(hr_string, channels=3)
  hr_img = tf.image.convert_image_dtype(hr_decoded, tf.float32)
  return lr_img, hr_img
```

### *3.2. Generate Super-Resolution (SR) images*

Before putting the validation data to the model, the weights of the networks must be restored. The graph, loss function and saver are still defined similar in the training process.

```
pred = sisr_cnn(X, scale, is_train=False, reuse=False)
loss = tf.losses.absolute_difference(pred, Y)
saver = tf.train.Saver(tf.global_variables())
```

The validation process is presented as below:

- Initialize all variables
- Restore the weights of the neural network
- Read the validation LR and HR image
- Feed LR image to the graph to generate SR image
- Save generated SR image
- Calculate L1 loss between generated SR image and the corresponding HR image
- Calculate PSNR and SSIM

```
with tf.Session() as sess:
   # Initialize variables
   sess.run(tf.global_variables_initializer())

   # Restore model weights from previous saved model
   saver.restore(sess, model_dir)
   ...
   valid_start = time.time()
   avg_loss = 0
   for i in range(n_valid):
      valid_img_start = time.time()
      lr_image, hr_image = sess.run([lr_img, hr_img])
      hr_output, loss_val = sess.run([pred, loss], feed_dict={X:lr_image,
Y:hr_image})
      avg_loss += loss_val
      save_image(hr_output, hr_gen, i+1)
```

Theoretically, the pixels in the output of the model can be any real values without lower and upper bound. However, the input LR patches are clipped values from 0 to 1, hence there are some output pixel values which are greater than 1 and less than 0 just little bit after floating point. Therefore, before save images, the pixel values must be multiplied with 255 then clipped in range [0, 255].

```
def save_image(hr, hr_dir, id):
  _, h, w, c = hr.shape
  hr = np.reshape(hr, [h,w,c])
```

```
hr = hr * 255.0
np.clip(hr, 0, 255, out=hr)
hr = hr.astype('uint8')
hr_save = Image.fromarray(hr)
hr_save.save(os.path.join(hr_dir, '{:04d}.png'.format(id)))
```

## 4. The experimental results

The neural network is trained and validate in the computer with an Intel(R) Core(TM) i7 @ 4.20GHz and a GPU NVIDIA GTX GeForce 960Ti. Total training time is about 7.6 hours and total validation time is about 27 seconds. In the training, the average loss of the final epoch is 0.221 while in the validation, the average loss is 0.208.

Table 4.1 shows the performance of my model and compare with Bicubic algorithm in Matlab about PSNR and SSIM.

**Table 4.1.** The comparison about performance between Bicubic algorithm and my network

| **Image** | **Bicubic** | | **My model** | |
|---|---|---|---|---|
| | **PSNR (dB)** | **SSIM** | **PSNR (dB)** | **SSIM** |
| 0015 | 24.01 | 0.8164 | 27.08 | 0.8768 |
| 0035 | 29.59 | 0.9125 | 32.67 | 0.9367 |
| 0055 | 28.47 | 0.8820 | 29.96 | 0.9018 |
| 0075 | 21.44 | 0.7903 | 22.81 | 0.8589 |
| 0095 | 30.68 | 0.9301 | 33.06 | 0.9184 |
| **Average** | 27.49 | 0.8796 | 29.91 | 0.8761 |

It is noticeable to see that the SISR network interpolates SR images better than Bicubic algorithm in term of PSRN and SSIM. The only drawback is in SSIM of image 0095.

However, some small details are not clear as in the original HR images. For example, the human stands far away in the image 0055 and the trees in the image 0075.


**Figure 4.1a.** The validation HR image 0015


**Figure 4.1b.** The generated SR image 0015

**Figure 4.2a.** The validation HR image 0035

**Figure 4.2b.** The generated SR image 0035

**Figure 4.3a.** The validation HR image 0055

**Figure 4.3b.** The generated SR image 0055

**Figure 4.4a.** The validation HR image 0075

**Figure 4.4b.** The generated SR image 0075

**Figure 4.5a.** The validation HR image 0095

**Figure 4.5b.** The generated SR image 0095