Name(s): Vhaijaiyanthishree Venkataramanan, Sridevi Divya Krishna Devisetty, Piyusha Jaisinghani
Instructor: Dr. Gita Sukthankar
Course Title: Machine Learning
Submission Date: November 27, 2018

Report - Super Resolution using CNNs

A technique which is used to reconstruct higher-resolution images from observed lower-resolution images is known as Super Resolution.

DATASET AND PREPROCESSING DETAILS

When User provides input image and desired resolution:

- The input contains path of the input image(jpg/png) and desired resolution.
- We create blur and sharp training data by upsampling and downsampling the input image upto 7 images and carry out training until specified epochs.
- After all the epochs are complete for the input_image, we take final output and upsample it by 1 pixel and train our cnn on this upsampled image, by creating train data as above.
- We repeat the above process until the output image of desired resolution is achieved.

When User has not specified any input data:

- The input is in the form of an RGB/Grayscale image(s) of dimension N X N and this image was downscaled to 500 x 500 pixels
- We then resized the 500 x 500 pixel images to 198 x 198 pixel sharp images and then upscaled it to 200 x 200 pixels to create a blurred image of the corresponding sharp image.
- This process was continued in intervals of 2 until we acquired the blurred version of the 500 x 500 image.

Image Name	# Sharp Images	# Blurred Images	Dimension Range
Image1	151	151	(200, 200, 3) ... (500, 500, 3)
Image2	151	151	(200, 200, 3) ... (500, 500, 3)
Image3	151	151	(200, 200, 3) ... (500, 500, 3)
Image4	151	151	(200, 200, 3) ... (500, 500, 3)

Table 1: Dataset details

- The above dimensions mentioned in Table 1 are with respect to RGB Images.
- Our training set consists of shuffled data with input as blurred images and output/ground truth as the difference of sharp and blurred images.

HYPER PARAMETER AND PARAMETER DETAILS

HYPERPARMETERS:

- **Number of Epochs:** 1000
- **Learning Rate:** 0.0001

PARAMETER INITIALIZATION:

- **Weights:**

We initialized our kernel weights using the He-et-al initialization

```
kernel_ size = kernel_height * kernel_width * input_channel

w = np.random.randn(kernel_height, kernel_width, input_channel,
output_channel)*np.sqrt(2/kernel_size)
```

Where, the kernel_size indicates the size of the filters and w indicates the weights matrix.

- **Biases:**

We initialized our biases to zero

```
b = np.zeros((1, 1, 1, output_channel))
```

Where, b indicates the bias matrix.

Model Architecture

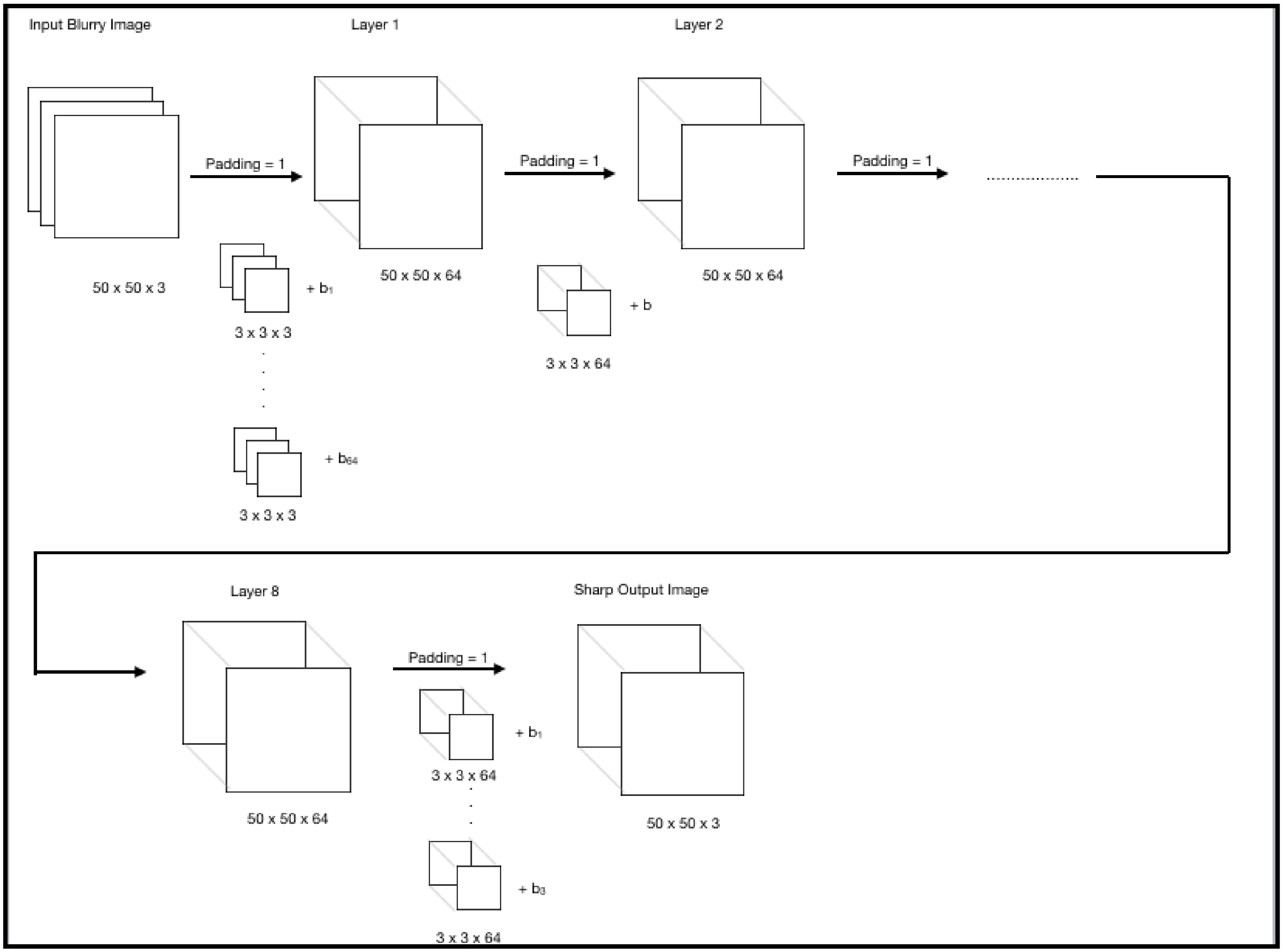


Fig. Network Architecture for Super Resolution using Convolutional Neural Networks

Note: The # of channels will vary with the type of input image (RGB = 3, Grayscale = 1)

Input Layer:

- Our input layer consists of an image with dimensions (Image_height, Image_width, # of Channels)
- This input image is convolved with 64 kernels with dimensions (3 x 3 x # of Channels)
- Each kernel has its own bias thereby making the number of biases 64 with dimensions (1, 1, 1, 64)

Hidden Layers:

- Our model consists of 8 hidden layers where each layer is a Convolution + ReLU Layer
- The nonlinear ReLU Layer is applied after each convolution layer in order to increase the training speed and also to alleviate the problem of vanishing and exploding gradients.

Output Layer:

- The output layer consists of the sharp image whose dimensions are the same as that of the input image (Image_height, Image_width, # of Channels)

CODE WALKTHROUGH

DATA PREPARATION:

```
def main(train_image, output_resolution):
    train_obj = train()
    type = train_image.rsplit(".")[1]
    if train_image is None or output_resolution is None:
        image_names_list = ['image_1.jpg', 'image_2.jpg', 'image_3.jpg', 'image_4.jpg']
        params, img = train_obj.train_cnn(image_names_list)
    else:
        params, img = train_obj.train_cnn([train_image])
        h, w, n_c = img.shape
        while h < output_resolution:
            scaled_image = cv2.resize(img, (h + 1, w + 1), interpolation=cv2.INTER_CUBIC)
            params, img = train_obj.train_cnn(scaled_image)
            h, w, n_c = img.shape
    output_file_name = train_image + "_sr" + str(output_resolution) + "." + type
    cv2.imwrite(output_file_name, img)
    with open('params.pkl', 'wb') as file:
        pickle.dump(params, file)
```

Our program accepts both jpg and png images.

If input image and desired resolution is given:

- We take the image and retrieve the image format.
- We create train data for the given input image as explained in section *dataset and preprocessing details*.
- Once the desired resolution image is achieved, we save the image with the same image format.
- To enable testing we saved our tuned optimal parameters in a pickle file.

Default input data:

- Here we consider multiple images, so we have considered a common downscaling constant as 500, irrespective of image size.
- Initially, all the input images are downsampled to 500 x 500 pixels.

- To generate the training set consisting of sharp images, we downsample the 500 x 500 images until we reached a pixel size of 200 x 200.
- Then each of the downsampled images were upsampled in order to generate the corresponding blurred images.
- Our training data consists of the blurred image as the input and the difference between sharp and blurred images as the ground truth value.
- To perform upsampling and downsampling we used opencv's cv2.resize function along with the bicubic interpolation parameter.

HYPERPARAMETER CONSTANTS:

```
learning_rate = 0.0001
epochs = 1000
```

- We chose the above mentioned hyper parameters to train our model.
- The number of epochs can be increased more in order to achieve better output.

WEIGHT AND BIAS INITIALIZATION:

```
def init_weights(filter_size, ip_layer, op_layer):

    filter_height, filter_width = filter_size
    w = np.random.randn(filter_height, filter_width, ip_layer, op_layer) *
    np.sqrt(2 / (ip_layer*filter_width*filter_height))

    return w


def init_bias(op_layer):

    return np.zeros((1, 1, 1, op_layer))
```

```
w1 = init_weights((3, 3), 3, 64)
b1 = init_bias(64)
w2 = init_weights((3, 3), 64, 64)
b2 = init_bias(64)
w3 = init_weights((3, 3), 64, 64)
b3 = init_bias(64)
w4 = init_weights((3, 3), 64, 64)
b4 = init_bias(64)
w5 = init_weights((3, 3), 64, 64)
b5 = init_bias(64)
w6 = init_weights((3, 3), 64, 64)
b6 = init_bias(64)
w7 = init_weights((3, 3), 64, 64)
b7 = init_bias(64)
w8 = init_weights((3, 3), 64, 64)
b8 = init_bias(64)
w9 = init_weights((3, 3), 64, 3)
b9 = init_bias(3)
```

- We have initialized weights with he et al and bias as zeros initially.
- The above code snippet is a sample of initializes kernels and bias at each layer with required dimensions.

CONVOLUTION LAYERS:

```
class Convolution():

    def __init__(self, input_channels, output_channels, filter_size, padding):
        self.weights = self.init_weights((filter_size, filter_size),
                                          input_channels, output_channels)
        self.bias = self.init_bias(output_channels)
        self.padding = padding
        self.params=(self.weights,self.bias)
        self.A_prev = None
```

The code snippet is a constructor for the class convolution. It initializes weights, biases, padding.

Forward propagation:

```python
def conv_forward(A_prev, W, b, hparameters):
    (n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = W.shape
    stride = hparameters['stride']
    pad = hparameters['pad']
    n_H = int((n_H_prev - f + 2 * pad) / stride) + 1
    n_W = int((n_W_prev - f + 2 * pad) / stride) + 1
    Z = np.zeros((n_H, n_W, n_C))
    A_prev_pad = zero_pad(A_prev, pad)
    for h in range(n_H):  # Looping over height of the image
        for w in range(n_W):  # Looping over width of the image
            for c in range(n_C):  # Looping over the number of kernels
                vert_start = h * stride
                vert_end = vert_start + f
                horiz_start = w * stride
                horiz_end = horiz_start + f
                a_slice_prev = A_prev_pad[vert_start:vert_end,
                                          horiz_start:horiz_end, :]
                Z[h, w, c] = conv_single_step(a_slice_prev, W[..., c],
                                              b[..., c])

    assert (Z.shape == (n_H, n_W, n_C))

    # Saving information in "cache" for the backprop
    cache = (A_prev, W, b, hparameters)

    return Z, cache
```

- At each layer, the forward convolution operation is carried out based on the above function.
- The below details are based on the assumption that an RGB image of size 500*500 is used and the following operations are performed based on the above method:

Input Layer Convolution

Image dimension at input - 500*500*3
Padded image dimensions - 502*502*3
Weight dimensions - filter height * filter width * channels * number of filters [3*3*3*64]
Bias dimensions - 1*1*1*64
Image dimension of the output of input layer convolution and ReLu = 500*500*64

Hidden Layer Convolution

Image dimension of input image to hidden layer - 500*500*64
Padded image dimensions - 502*502*64
Weight dimensions - filter height * filter width * channels * number of filters [3*3*64*64]
Bias dimensions - 1*1*1*64
Image dimension of the output of hidden layer convolution and ReLu= 500*500*64

Output Layer Convolution

Image dimension of input image to output layer - 500*500*64
Padded image dimensions - 502*502*64
Weight dimensions - filter height * filter width * channels * number of filters [3*3*64*3]
Bias dimensions - 1*1*1*64
Image dimension of the output of output layer convolution = 500*500*3

BACKWARD PROPAGATION:

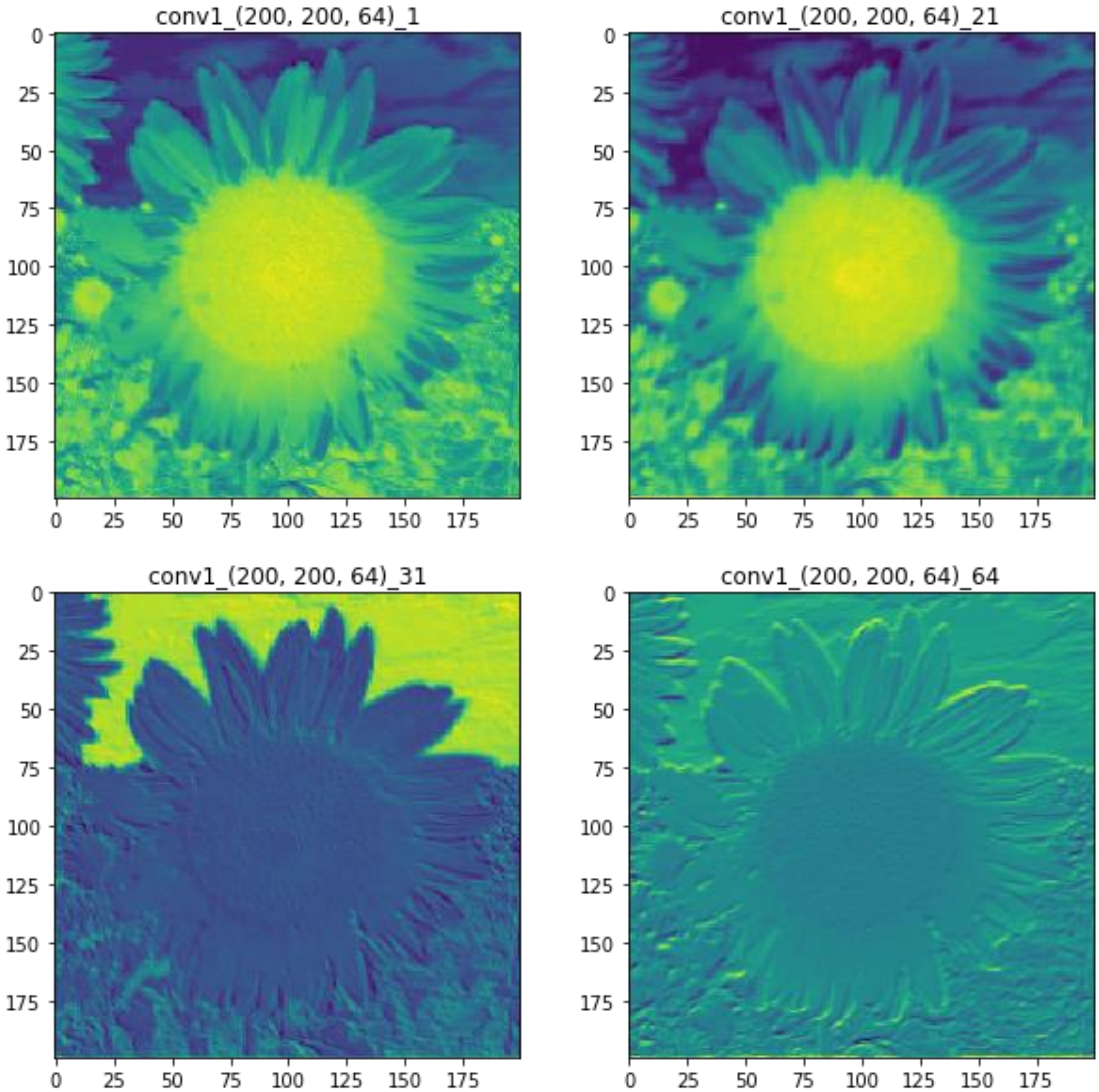
```
def conv_backward(self, dZ, cache):

    (n_H_prev, n_W_prev, n_C_prev) = self.A_prev.shape

    (f, f, n_C_prev, n_C) = self.weights.shape

    pad = self.padding

    (n_H, n_W, n_C) = dZ.shape

    dA_prev = np.zeros((n_H_prev, n_W_prev, n_C_prev))
    dW = np.zeros((f, f, n_C_prev, n_C))
    db = np.zeros((1, 1, 1, n_C))

    A_prev_pad = self.zero_pad(self.A_prev, pad)
    dA_prev_pad = self.zero_pad(dA_prev, pad)

    for h in range(n_H):  # Looping over the height of the image
        for w in range(n_W):  # Looping over the width of the image
            for c in range(n_C):  # Looping over the number of kernels

                a_slice = A_prev_pad[h:h + f, w:w + f, :]

                # Update gradients for the window and the filter's
                # parameters using the code formulas given above
                dA_prev_pad[h:h + f, w:w + f, :] +=
                self.weights[:, :, :, c] * dZ[h, w, c]
                dW[:, :, :, c] += a_slice * dZ[h, w, c]
                db[:, :, :, c] += dZ[h, w, c]

    dA_prev[:, :, :] = dA_prev_pad[pad:-pad, pad:-pad, :]

    # Making sure the output shape is correct
    assert (dA_prev.shape == (n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, (dW, db)
```

- To adjust the weights and biases we now perform backward propagation.

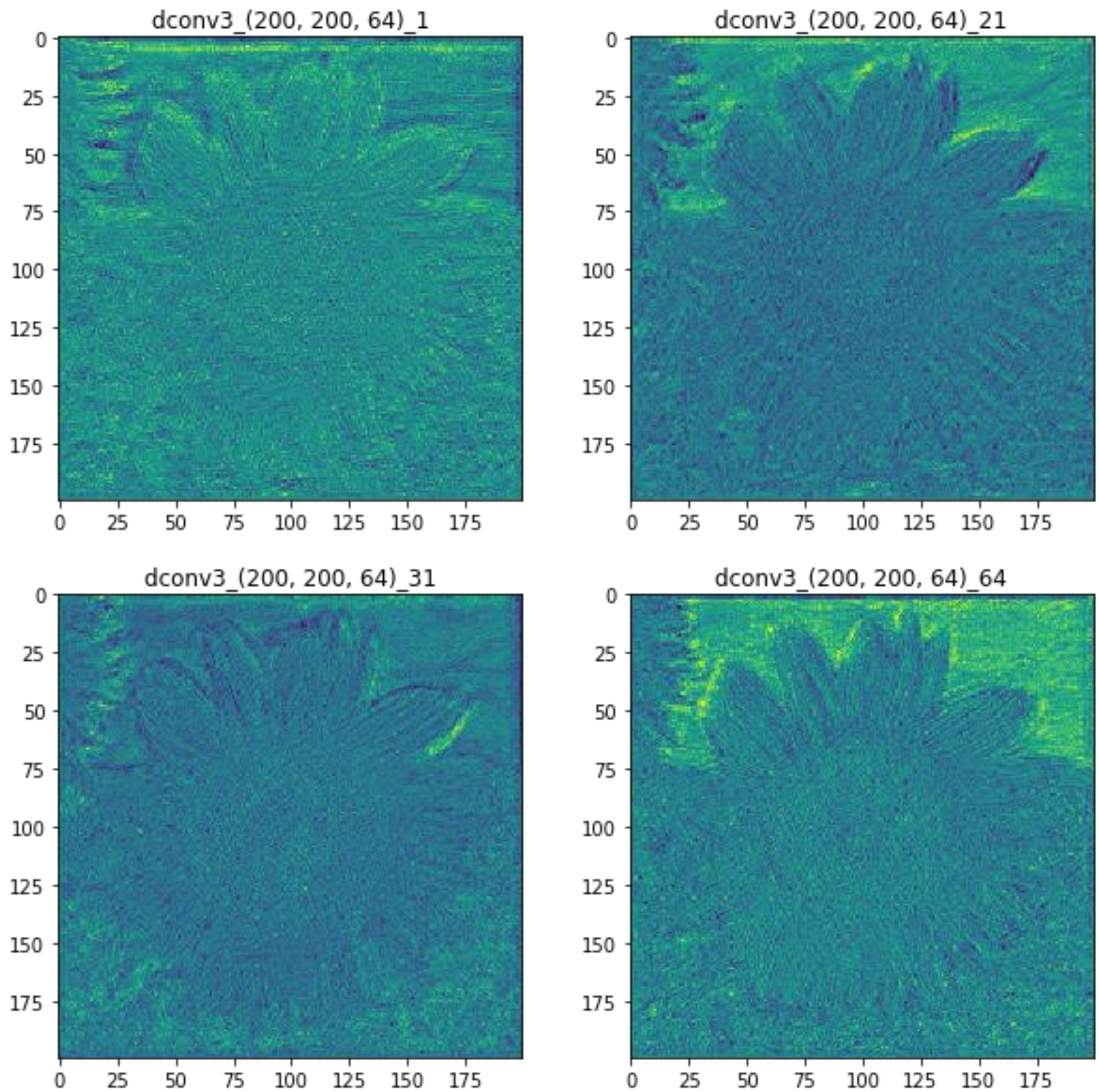
- The input image at the deconvolution layer is an image which is the difference between final convolution layer output and ground truth image, which is nothing but our loss gradient.

Test Case output for convolution layer with forward and backward function calls:

Forward Convolution output at epoch 3:



Backward convolution output at epoch 3:



ACTIVATION FUNCTION RELU:

Forward Propagation:

During forward propagation, each convolved output is fed to the relu_forward function as shown below.

Backward Propagation:

During backward propagation, the gradient of convolution is fed to the relu_backward function as shown below.

```
class ReLU():
    def __init__(self):
        self.X = None

    def relu_forward(self, X):
        """
        Computes the forward pass for a layer of rectified linear unit
        (ReLUs). Input:
        - x: Inputs, of any shape
        """
        self.X = X
        self.X[self.X <= 0] = 0
        return self.X

    def relu_backward(self, dout):
        """
         Computes the backward pass for a layer of rectified linear units
         (ReLUs). Input:
        """
        dX = dout.copy()
        dX[self.X <= 0] = 0
        return dX
```

LOSS FUNCTION (Mean Square Error):

```
class MSE:
    def mse(self, A, B):
        mse_loss = (np.square(np.subtract(A, B))).mean()
        mse_grad = A - B
        return mse_loss, mse_grad
```

- Mean square error of final convolution layer output that is at layer 9, and ground truth is calculated. The above method calculates both loss and loss gradient.

- Loss gradient is used as input for backpropagation.

TRAINING THE MODEL:

```
layer_1 = Convolution(ip_channels, out_channels, filter_size, padding)
relu_1 = ReLU()
layer_2 = Convolution(out_channels, out_channels, filter_size, padding)
relu_2 = ReLU()
layer_3 = Convolution(out_channels, out_channels, filter_size, padding)
relu_3 = ReLU()
layer_4 = Convolution(out_channels, out_channels, filter_size, padding)
relu_4 = ReLU()
layer_5 = Convolution(out_channels, out_channels, filter_size, padding)
relu_5 = ReLU()
layer_6 = Convolution(out_channels, out_channels, filter_size, padding)
relu_6 = ReLU()
layer_7 = Convolution(out_channels, out_channels, filter_size, padding)
relu_7 = ReLU()
layer_8 = Convolution(out_channels, out_channels, filter_size, padding)
relu_8 = ReLU()
layer_9 = Convolution(out_channels, ip_channels, filter_size, padding)
```

- We initialize all the convolution layers and Relu layers as above.
- We then pass all these 8 hidden layers to our CNN object, which invokes forward and back propagation on all the layers as follows:

```
class CNN:

    def __init__(self, layers, loss_func=MSE.mse):
        self.layers = layers
        self.params = []
        for layer in self.layers:
            self.params.append(layer[0].params)
        self.loss_func = loss_func

    def forward(self, X):
        for layer in self.layers:
            X = layer[0].conv_forward(X)
            print(X.shape)
            X = layer[1].relu_forward(X)
        return X

    def backward(self, dout):
        grads = []
        for layer in reversed(self.layers):
            dout, grad = layer[0].conv_backward(dout)
            if layer[1]:
                dout = layer[1].relu_backward(dout)
            grads.append(grad)
        return grads

    def train_step(self, X, y):
        out = self.forward(X)
        loss, dloss = self.loss_func(out, y)

        grads = self.backward(dloss)

        return loss, grads
```

The train_step method of CNN class, invokes forward propagation, computes loss and performs backward propagation on every image and returns loss and updated gradients.

Gradient update:

```
for epoch in range(epochs):
    for x, y in self.train_data:
        cnn_obj = CNN(layers)
        loss, grads = cnn_obj.train_step(x, y)
        print("loss:", loss)
        params1, params2, params3, params4, params5, params6,
        params7, params8, params9 = grads
        dw1, db1 = params1
        dw2, db2 = params2
        dw3, db3 = params3
        dw4, db4 = params4
        dw5, db5 = params5
        dw6, db6 = params6
        dw7, db7 = params7
        dw8, db8 = params8
        dw9, db9 = params9

        decay = lr * (epoch/epochs)
        lr = (lr * 1 / (1+ decay))
        layer_1.weights = layer_1.weights - (lr * dw1)
        layer_1.bias = layer_1.bias - (lr * db1)
        layer_2.weights = layer_2.weights - (lr * dw2)
        layer_2.bias = layer_2.bias - (lr * db2)
        layer_3.weights = layer_3.weights - (lr * dw3)
        layer_3.bias = layer_3.bias - (lr * db3)
        layer_4.weights = layer_4.weights - (lr * dw4)
        layer_4.bias = layer_4.bias - (lr * db4)
        layer_5.weights = layer_5.weights - (lr * dw5)
        layer_5.bias = layer_5.bias - (lr * db5)
        layer_6.weights = layer_6.weights - (lr * dw6)
        layer_6.bias = layer_6.bias - (lr * db6)
        layer_7.weights = layer_7.weights - (lr * dw7)
        layer_7.bias = layer_7.bias - (lr * db7)
        layer_8.weights = layer_8.weights - (lr * dw8)
        layer_8.bias = layer_8.bias - (lr * db8)
        layer_9.weights = layer_9.weights - (lr * dw9)
        layer_9.bias = layer_9.bias - (lr * db9)
```

- The above code runs for various epochs.
- In each epoch we run for every image in train data and perform convolution(forward+backward) operation.
- After each convolution, we adjust weights and biases based on the learning rate and gradients obtained from convolution operation.
- After all the epochs are completed, we save the optimal parameters in a pickle file, which is used for testing purpose . We also save the super resolution image received after training.

Super Resolution Algorithm:

```
def main(train_image, output_resolution):
    train_obj = train()
    type = train_image.rsplit(".")[1]
    if train_image is None or output_resolution is None:
        image_names_list = ['image_1.jpg', 'image_2.jpg', 'image_3.jpg', 'image_4.jpg']
        params, img = train_obj.train_cnn(image_names_list)
    else:
        params, img = train_obj.train_cnn([train_image])
        h, w, n_c = img.shape
        while h < output_resolution:
            scaled_image = cv2.resize(img, (h + 1, w + 1), interpolation=cv2.INTER_CUBIC)
            params, img = train_obj.train_cnn(scaled_image)
            h, w, n_c = img.shape
    output_file_name = train_image + "_sr" + str(output_resolution) + "." + type
    cv2.imwrite(output_file_name, img)
    with open('params.pkl', 'wb') as file:
        pickle.dump(params, file)
```

- The else part uses user provided image and executes super resolution algorithm.

```
def get_train_data(self, train_images):
    input_images = []
    X = []
    Y = []
    if len(train_images) == 1:
        input_img = mpimg.imread(train_images[0])
        h, w, n_c = input_img.shape
        input_image = cv2.resize(input_img, (h, w), interpolation=cv2.INTER_CUBIC)
        for i in range(8):
            Y.append(cv2.resize(input_image, (h - i, w - i), interpolation=cv2.INTER_CUBIC))
        print(Y[0].shape, Y[-1].shape)

        for j in range(1, len(Y)):
            h, w, _ = Y[j].shape
            X.append(cv2.resize(Y[j], (h + 1, w + 1)))
        print(X[0].shape, X[-1].shape)
```

- We create blur and sharp training data by upsampling and downsampling the input image upto 7 images and carry out training until specified epochs.

- For example if input image is 50x50x3, we create blur image by first resizing it to 49x49x3(sharp) using bicubic interpolation and then we create 50x50x3(blur). We follow the up/down sample same steps to create till 44x44x3 sharp images and then convert them to blur images.

- After all the epochs are complete for the input_image, we take final output and upsample it by 1 pixel and train our cnn on this upsampled image, by creating train data as above.

- We repeat the above process until the output image of desired resolution is achieved.

- We save the output image in the given input image format.(jpg/png) using cv2.imwrite function as shown in above code snippet.

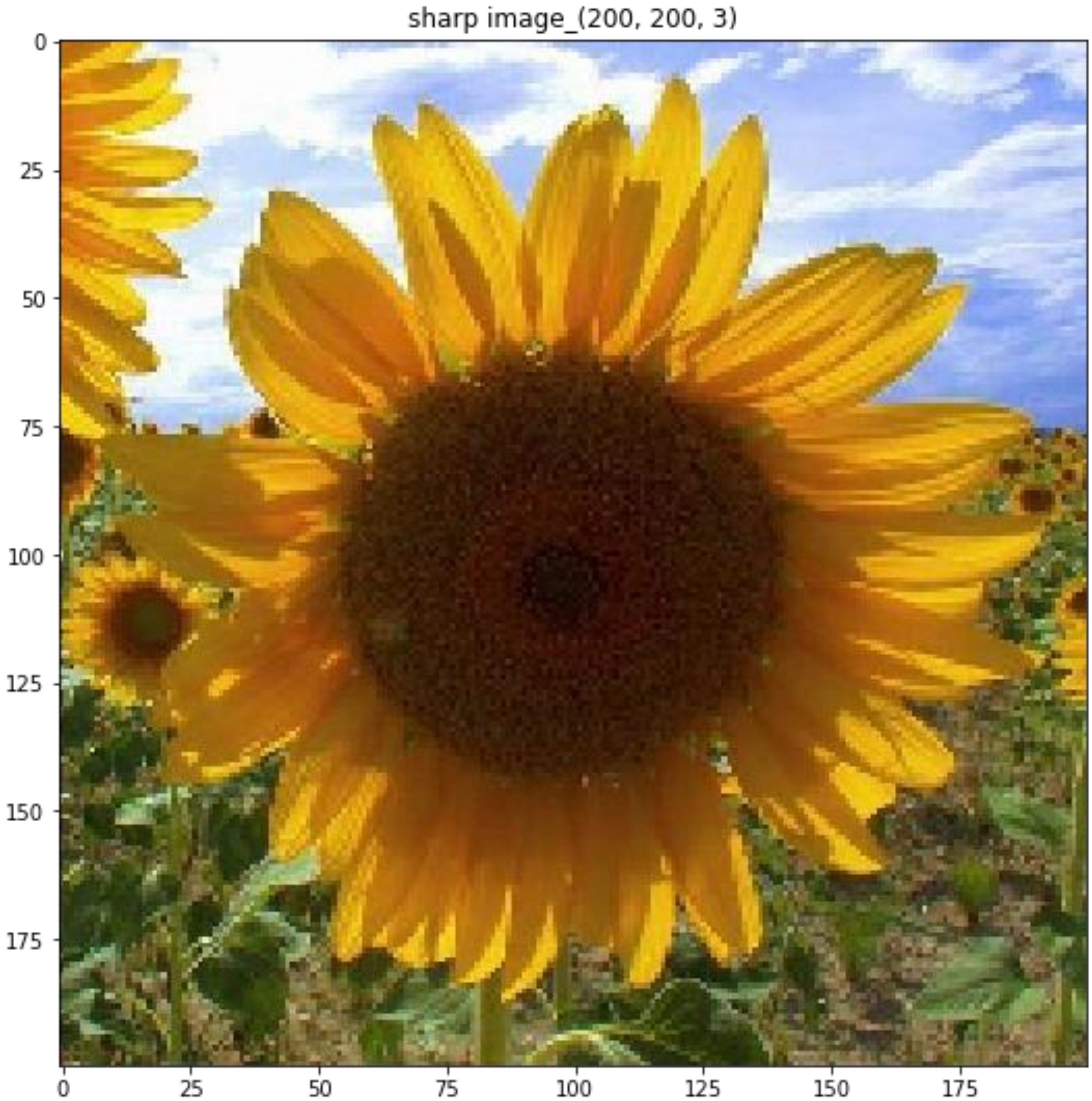
Sample Test code:

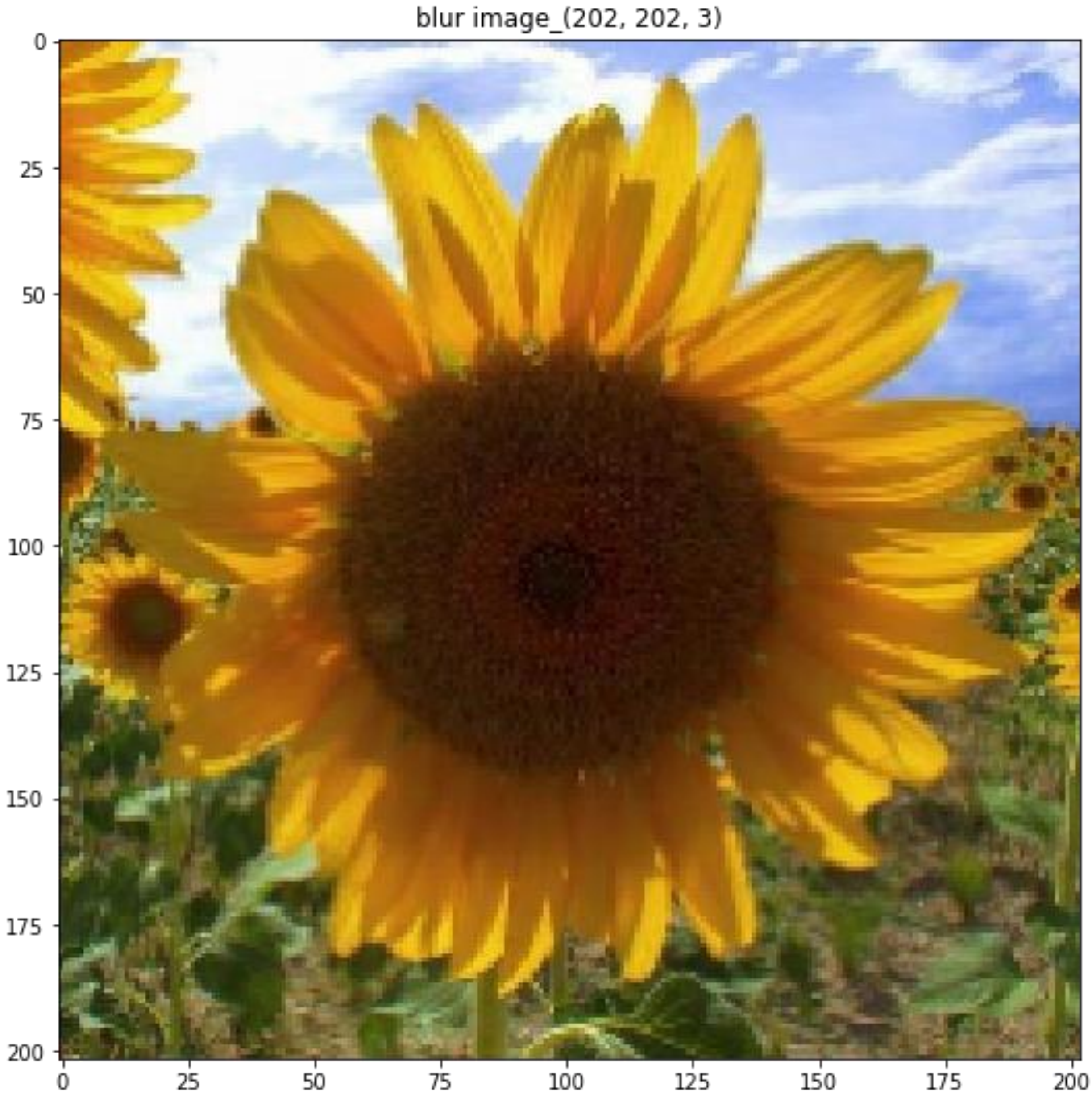
```
def test_cnn(test_image_name):
    test_img = mpimg.imread(test_image_name)
    pkl_file = open('params.pkl', 'rb')
    type = test_image_name.rsplit(".")[1]
    params = pickle.load(pkl_file)
    params1, params2, params3, params4, params5, params6, params7, params8, params9 = params
    h, w, ip_channels = test_img.shape
    out_channels = 64
    filter_size = 3
    padding = 1
    layer_1 = Convolution(ip_channels, out_channels, filter_size, padding)
    layer_1.weights = params1[0]
    layer_1.bias=params1[1]
    relu_1 = ReLU()
    layer_2 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_2.weights = params2[0]
    layer_2.bias = params2[1]
    relu_2 = ReLU()
    layer_3 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_3.weights = params3[0]
    layer_3.bias = params3[1]
    relu_3 = ReLU()
    layer_4 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_4.weights = params4[0]
    layer_4.bias = params4[1]
    relu_4 = ReLU()
    layer_5 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_5.weights = params5[0]
    layer_5.bias = params5[1]
    relu_5 = ReLU()
    layer_6 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_6.weights = params6[0]
    layer_6.bias = params6[1]
    relu_6 = ReLU()
    layer_7 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_7.weights = params7[0]
    layer_7.bias = params7[1]
    relu_7 = ReLU()
    layer_8 = Convolution(out_channels, out_channels, filter_size, padding)
    layer_8.weights = params8[0]
    layer_8.bias = params8[1]
    relu_8 = ReLU()
    layer_9 = Convolution(out_channels, ip_channels, filter_size, padding)
    layer_9.weights = params9[0]
    layer_9.bias = params9[1]
    layers = [(layer_1, relu_1), (layer_2, relu_2), (layer_3, relu_3), (layer_4, relu_4), (layer_5, relu_5),
              (layer_6, relu_6),
              (layer_7, relu_7), (layer_8, relu_8), (layer_9, None)]
    cnn_obj = CNN(layers)
    out_img = cnn_obj.forward(test_img)
    output_file_name = test_image_name + "_sr" + str(h) + "." + type
    cv2.imwrite(output_file_name, out_img)
```

- We can invoke this method for testing the model.
- It will initialize the model with weights and biases read from the pickle file.
- It will produce super resoluted output image and saves it in the same image format.

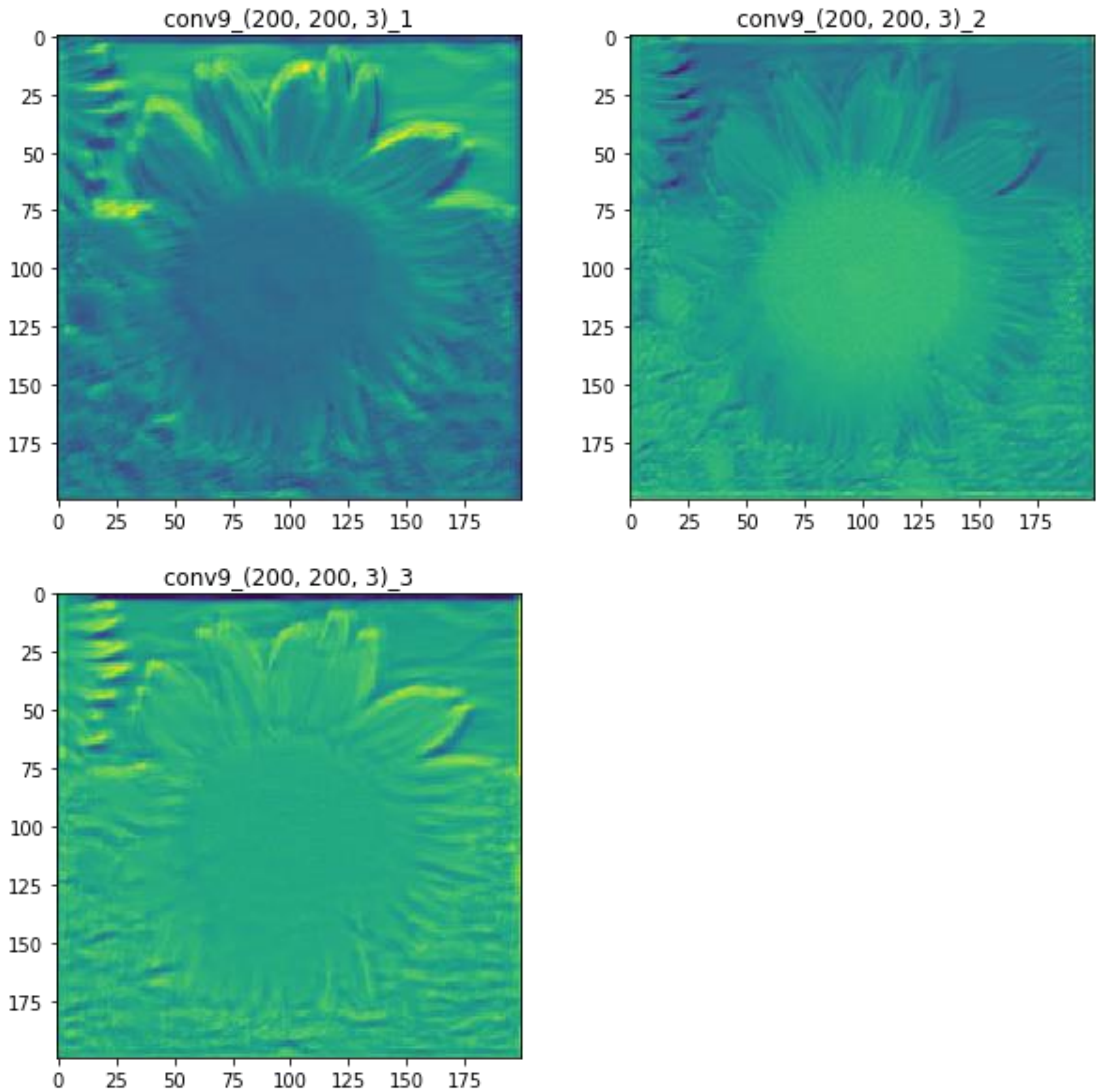
Results:

Downsampled to 200x200(sharp) and upsampled to get 202x202 blur image:


sharp image_(200, 200, 3)
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175


blur image_(202, 202, 3)
0
25
50
75
100
125
150
175
200
0
25
50
75
100
125
150
175
200

Convolution output at 250th epoch:



The above image is of dimension 200x200x3. Each image is with respect to individual channel.

Note:

We could not run for more epochs as the processing time was too high with our default data inputs. With small input data dimensions we observed that our cost was fluctuating a lot.

Source code structure:

Main.py : implements superresolution algorithm. It also contains code for data preparation and invoking CNN.

Cnn.py : It contains layers and its parameters. Contains forward and backward methods to invoke on these layers(hold layers of both convolution and relu).

layers.py : It holds weights and biases w.r.t to its layer. It performs forward and backward convolution.

Relu.py : It performs forward and backward activation.

Loss.py : It contains Mean square error method to calculate loss and its gradient.

Also attached cumulative code named CNN which has all methods in the same class.