Guide for DeepL srSMLM

by Hanna MANKO
code is available on GitHub

Contents

1 Patches preparation

First we need to import all the required libraries:

```
import numpy as np
import pandas as pd
from tifffile import imread, imsave
from sklearn.cluster import DBSCAN
from matplotlib import pyplot as plt
from numpy import unique
from numpy import where
import time
import math
import random
from skimage.morphology import disk
from scipy.ndimage.morphology import white_tophat
from joblib import Parallel, delayed
```

Then we define the important variables as size of the patch and the distance between boxes of positional and spectral peaks

```
### Defining Patches Size
image_height = 16  ## image height in pixels
loc_image_width = 16  ## image width in pixels
image_chanels = 1   ##
loc_sp_distance = 230  ## distance between localization and its spectral peak
                        #   measured from original image / in pixels
sp_image_width = 48 #### spectral image width
half_width = int(loc_image_width/2)    ##
```

Now there is quite a big part where we define all the functions which will be used during patches preparation process. Here, you can control the shifts which will be introduced into localization and spectral parts (The shifts are required during training procedure to avoid the network learning specific positions of signal in the box):

```
##################### DEFINING THE  FUNCTIONS ###################
###
##### #######    Useful  function to plot on one figure multiple patchs
def show_images(patches):
    n = int(math.sqrt(len(patches))+2)
    m = int(math.sqrt(len(patches)))
    plt.figure(figsize = (50,40))
    for i in range (0, len(patches)):
        plt.subplot(n,m,i+1)
        plt.title(i, fontsize=25)
        plt.imshow(patches[i])
    plt.show()

#######
def set_coordinates(i, half_width, loc_sp_distance, sp_image_width):
    row = np.array(where(cluster == i))
    ## coordinates for localization part
    loc_sp_distance = loc_sp_distance+random.randint(-10,20)
    shift_y = random.randint(-6,6)   ##    shift for x and y coordinates. To move the box around
                                                   localization
    shift_x = random.randint(-6,6)   ##
   # shift_y = 0   ##      if shift is not requires. When we need to create patches
   # shift_x = 0   ##
    y1 = int(dat[np.array(row)[0,0],0]-half_width-shift_y)   #   start y-coordinate of the box
                                                   around localization
    y2 = int(dat[np.array(row)[0,0],0]+half_width-shift_y)   #   end y-coordinate of the box around
                                                   localization
    x1 = int(dat[np.array(row)[0,0],1]-half_width-shift_x)   #   start x-coordinate of the box
                                                   around localization
    x2 = int(dat[np.array(row)[0,0],1]+half_width-shift_x)   #   end y-coordinate of the box around
                                                   localization
    ## coordinates for spectral part
    x = dat[np.array(row)[0,0],0]
    y = dat[np.array(row)[0,0],1]
    y11 = int(dat[np.array(row)[0,0],0]+loc_sp_distance-shift_y)    #   start y-coordinate of the
                                                   box around spectra
    y21 = int(dat[np.array(row)[0,0],0]+loc_sp_distance + sp_image_width-shift_y) #   end y-
                                                   coordinate of the box around spectra
    return(y1,y2,x1,x2,y11,y21,row,x,y)

####   The code was first writen to be able to create patches of different types
##   Function that returns image shape depending on patch type chosen
def image_type(image_height, sp_image_width,loc_image_width):
    if only_loc == True:                                        ##  to have only localization part
        im_shape = (image_height, loc_image_width)
    if only_spectra == True:                                    ##  to have only spectral part
        im_shape = (image_height, sp_image_width)
    if combined == True:                                        ##  to have both localization and
                                                       spectral parts
        im_shape = (image_height, sp_image_width+loc_image_width)
    return (im_shape)

#   Function that forms the patches depending on patch type chosen
sp_cor = 0   ###  the number of pixels in x-axis to corect position of the box around spectral part
def patches_formation(row, stack):
    sp_cor = random.randint(-1,1)  ##    to introduce random shift for the box around spectral part
    frame = dat[:,2][row]
    if only_loc == True:
        patch = np.float64(stack[frame, x1:x2, y1:y2])
        patch = patch.reshape(row.shape[1],16,16)
    if only_spectra == True:
        patch = np.float64(stack[frame, x1:x2, y11:y21])
        patch = patch.reshape(row.shape[1],16,48)
    if combined == True:
        patch = np.concatenate((np.float64(stack[frame, x1:x2, y1:y2]),
                                np.float64(stack[frame, x1-sp_cor:x2-sp_cor, y11:y21])), axis=3)
        patch = patch.reshape(row.shape[1],16,64)
    return(patch)

#######   Function to calculate Spatial Frecuency as described in (Shutao Li et al.
# 'Combination of images with diverse focuses using the spatial frequency',  Information Fusion
# https://doi.org/10.1016/S1566-2535(01)00038-0.
def SF_calculator(patch):
    MN = image_height*sp_image_width
    rf = np.sqrt((1/MN)*np.sum([np.abs((patch[:,n]-patch[:,n-1])**2) for n in range(1,sp_image_width
                                                       -1)]))
    cf = np.sqrt((1/MN)*np.sum([np.abs((patch[m,:]-patch[m-1,:])**2) for m in range(1,image_height-1
                                                       )]))
    SF = np.sqrt(rf**2 + cf**2)
    return SF

# Two functions to denoise spectral and localization  part
selem = disk(10)
def d_spectra(img, tempIm, ampFact = .5):   ## ampFact is introduced to increase signal for the
                                                   spectral part
    imgMean1 = tempIm[0, 0:img.shape[2]]/(ampFact*tempIm[0:img.shape[1],0:img.shape[2]].max())
    imgMean1 = white_tophat(imgMean1, footprint =  selem)
    return(imgMean1)
def d_loc(img, tempIm):
    imgMean2 = tempIm[0, 0:img.shape[2]]/tempIm[0:img.shape[1],0:img.shape[2]].max()
    imgMean2 = white_tophat(imgMean2, footprint = selem)
    return(imgMean2)

#######    /// Spatial Frequency calculation and image denoisin  \\\
def SF_Image(patch, only_loc=False, only_spectra=False, combined=False, denoising = True):
    SF=[]
    SF_sum = 0
    cumIm = np.zeros((patch[1:].shape))
    for i in range(0,len(patch[1:])):
        SF.append(SF_calculator(patch[i]))
        cumIm = cumIm + patch[i]*SF[i]
    SF_sum = np.sum(SF)
    tempIm = cumIm/SF_sum
    if denoising ==True:
        if only_spectra == True:
            imgMean = d_spectra(patch[:,:,loc_image_width:], tempIm[:,:,loc_image_width:])
        elif only_loc == True:
            imgMean = d_loc(patch[:,:,0:loc_image_width], tempIm[:,:,0:loc_image_width])
        elif combined == True:
            imgMean = np.concatenate((d_loc(patch[:,:,0:loc_image_width],
                  tempIm[:,:,0:loc_image_width]),(d_spectra(patch[:,:,loc_image_width:],
                           tempIm[:,:,loc_image_width:]))),axis=1)
        return(imgMean)
    else:
        return(tempIm)
####
#   function that exclude one frame and apply SF_Image function to all othe frames
def SF_part(n, patch_x, pat_s):
    fr = [*range(0,len(patch_x))]
    fr.pop(n)
    pat_s = SF_Image(patch_x[((fr)),:,:], combined=True)
    return (pat_s)
```

Now we can start the data preparation

Here we are using the localizations file obtained from the PeakFit Fiji plugin. In this case before uploading it is required to delete first 7 rows and # before Frame and save the file in *.csv format (this also can be done directly in python). It is also possible to use thunderSTORM files which usually provides the localizations in nm so you will need to transform the data back into the pixels (this step is dependant on the pixel size of the specific setup).

```
############################################################### 3
############################################################### 2
############################################################### 1 ...
##########################    ## Put the type of images that you want to obtain equal True
###///                          Here we will create images containing both spectral and localization
                                                   parts
only_spectra = False
only_loc = False
combined = True
###\\\
##########################
####    localization file from PeakFit (part of GDSC SMLM2 plugin in Fiji)  ###

markerSize = 0.05

data = data[data["Signal"]>5000]  ## it is preferable to filter data by signal intensity (see Figure
                                                   1)
plt.plot(data["origX"], data["origY" ], 'o', markersize=markerSize, color='red') ## we can have a
                                                  look at the scatted plot
datat = pd.concat([pd.DataFrame(data["origX"]),pd.DataFrame(data["origY"]),pd.DataFrame(data["Frame"
                                                  ])], axis = 1, ignore_index=False)

data_n= datat.to_numpy()
d = data_n[(data_n[:,0]>10)&(data_n[:,0]<220)&(data_n[:,1]>10)&(data_n[:,1]<250)] ## cropping the
                                                  field of view to work only with localizations part
plt.plot(d[:,0], d[:,1], 'o', markersize=markerSize, color='red')

data = data[(data['origX']>10)&(data['origX']<220)&(data['origY']>10)&(data['origY']<250)]
data = data.reset_index()
```

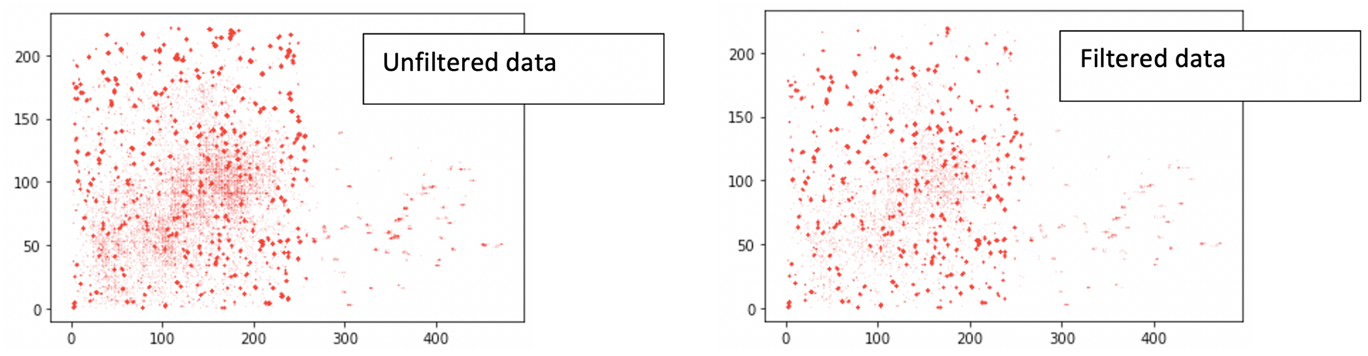

Figure 1: The unfiltered data and data which was filtered by signal intensity

Next important step is localizations clustering. For this we are using DBSCAN. eps and min_samples are adjusted depending on the data

```
##############################################################################
```

```
#####       Next we cluster the localizations on consecutive frames to create 'clean' images from
                                                   the same localization
##          For this we are using DBSCAN

cl = DBSCAN(eps=0.001,min_samples=3)    # clustering   ## eps need to be chosed depending on the
                                                   data (see line)
cluster = cl.fit_predict(d[:, 0:2])   # table with indexes of all clusters
clusters = unique(cluster)            # tacking unique indexes of clusters
clusters = clusters[clusters>-1]      # discarding noise
#---
def consecutive(data, stepsize=1):             # small function that helps to find consecutive
                                                   frames for cluster
    return np.split(data, np.where(np.diff(data) != stepsize)[0]+1)
#---
dat = pd.concat([pd.DataFrame(d), pd.DataFrame(cluster)], axis = 1)  # Adding cluster indexes as
                                                   additional column to our data
dat = dat.to_numpy()  # Converting to numpy

cluster_2 = np.int64(np.full((cluster.shape), -1))  # Creation array full of '-1'
new_cluster = 0
for i in clusters:
    ar = consecutive(dat[:,2][(where(dat[:,3] == i))])  # geting consecutive indexes in data for
                                                       each cluster
    for j in ar:                # for ezch of the consecutive index sets
        if len(j) > 8:          # if this set is longer then 8
            for ii in range(0, len(j)):
                cluster_2[where((dat[:,2]==j[ii])&(dat[:,3] == i))] = int(new_cluster)  # put new
                                                                   cluster index to cluster_2
            print(new_cluster)
            new_cluster = new_cluster+1  # increase new_cluster index by 1

clusters_2 = unique(cluster_2)  #  again finding unique clusters
clusters_2 = clusters_2[clusters_2>-1]  # discarding the noise
dat[:,3] = cluster_2   #  put cluster indexes as column 3 in data
clusters = clusters_2  #  rewrite clusters
cluster = cluster_2
del cluster_2, clusters_2  # deleting additional variables

for clust in clusters:          ##  Using this loop we can plot data by coloring different clusters
  row_ix = where(cluster == clust)
  plt.scatter(dat[row_ix, 0], dat[row_ix, 1], s=markerSize)
plt.show()

data['clusterID'] = pd.DataFrame(cluster)
```

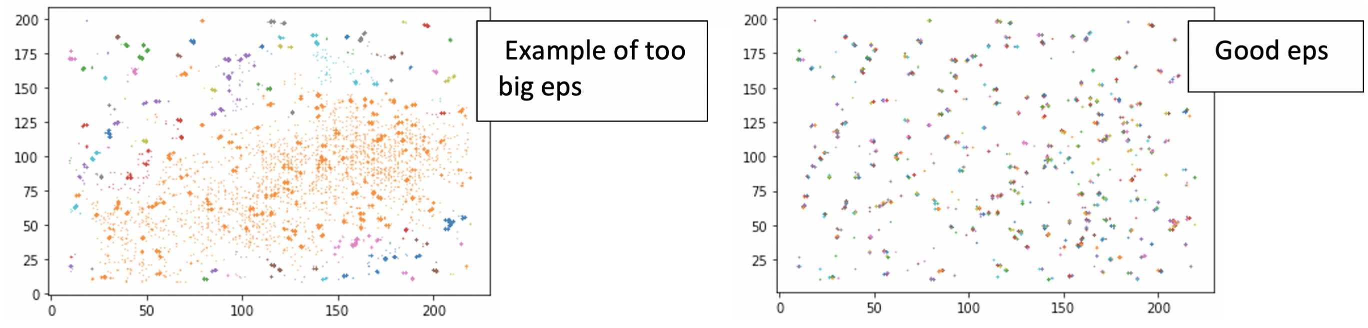

Figure 2: Examples of cluster images obtained with different eps.

Now you can see the main loop for patches creation. Here first you need to upload the raw image stack and normalize it. In this loop we are using parallel tasks, otherwise the process may take significant amount of time.

```
stack = imread("/path/25mW_2_MMStack_Pos0.ome.tif")
stack = stack/2**16 ####   Normalization

start_time = time.time()
```

```
print("--- %s seconds ---" ,(time.time() - start_time))   # printing start time
### Creating the training set
##########
im_shape = image_type(image_height, sp_image_width,loc_image_width)   # Reading image shape
count = 0
print("Please, wait until the end")
start_time = time.time()
x1_list = []
y1_list = []
SF_p_list = []
SF_sum_list = []
data_v = pd.DataFrame()
for i in clusters:
    y1,y2,x1,x2,y11,y21,row,x,y = set_coordinates(i,half_width,loc_sp_distance, sp_image_width)  #
                                                    defining the cordinates of the box
    patch_x = np.zeros((np.shape(row[0])[0], im_shape[0], im_shape[1]))  # creating the empty patch
                                                    of defined size
    try:
        patch = patches_formation(row, stack)
    except:
        continue
    patch_x = patch
    pat_s = np.zeros(patch_x.shape)
    SF_p = []
    SF_sum = []
    pat_s = Parallel(n_jobs=8)(delayed(SF_part)(n, patch_x, pat_s) for n in range(0, len(patch_x)))
    # creating 'clean' images
    pat_s = np.array(pat_s).reshape(patch_x.shape)  # reshaping
    SF_sum = [SF_calculator(pat_s[n]) for n in range(0, len(pat_s))]
    # saving calculated SF value
    SF_p = [SF_calculator(patch_x[n]) for n in range(0, len(patch_x))]
    # calculating SF value for one excluded pacth
    ind = where(np.array(SF_sum)/np.array(SF_p) > ((np.array(SF_sum)/np.array(SF_p)).max()-
                 (np.array(SF_sum)/np.array(SF_p)).min())/2)
    patch_ = patch_x[ind]
    patch_sum  = pat_s[ind]
    x1_list.extend([x]*len(ind[0]))
    SF_p_list.extend(np.array(SF_p)[ind[0]])
    SF_sum_list.extend(np.array(SF_sum)[ind[0]])
    y1_list.extend([y]*len(ind[0]))
    data_v = data_v.append(data.iloc[list(row[0][np.array(ind)][0])], ignore_index = True)
    if count == 0:            #   gethering all the patches in one big set
        patch__ = patch_      #   set of original images
        patch_sum_  = patch_sum       #   set of created 'clean' images
    else:
        patch__ = np.concatenate((patch__, patch_), axis=0)  #   set of original images
        patch_sum_ = np.concatenate((patch_sum_, patch_sum), axis=0)   #   set of created 'clean'
                                                        images
    count = count+1
    print(count)
print("--- %s seconds ---" % (time.time() - start_time))    #  printing  end time



patch__v = patch__.reshape(len(patch__),64*16)
patch_sum_v = patch_sum_.reshape(len(patch_sum_),64*16)

data_vv = pd.concat((data_v,pd.DataFrame(patch__v), pd.DataFrame(patch_sum_v)), axis = 1)

ind = []
for i in range(0, len(patch__)):
    if patch__[i,:16,:].max() <0.09:
        ind.append(i)
patch___ = np.delete(patch__, [ind], axis=0)
patch_sum__ = np.delete(patch_sum_, [ind], axis=0)
data_vv = data_vv.drop(data_vv.index[ind])

data_vv.to_csv('/path/data_150_2.csv', sep = ';', decimal='.', index = False,encoding="utf-8")
imsave('/path/150_2_patch_sum.tif', patch_sum__)
imsave('/path/150_2_patch.tif', patch___)
```

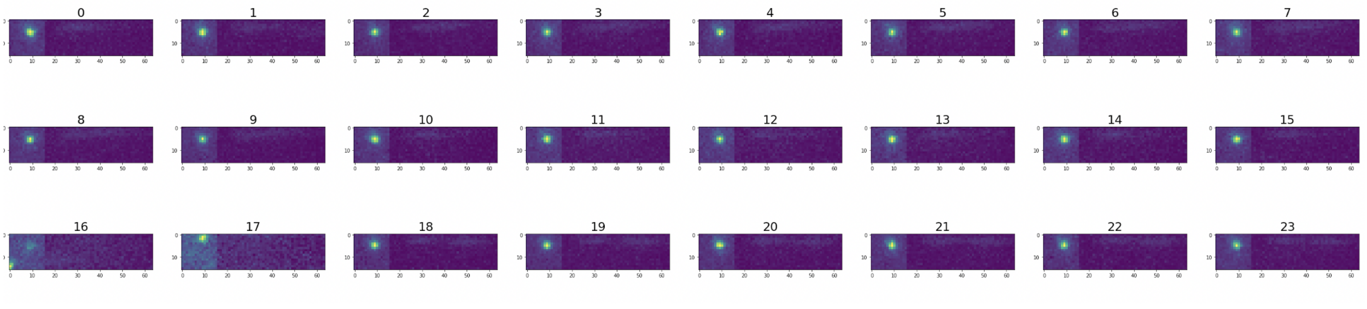

Figure 3: Raw low-SNR patches (patch___)

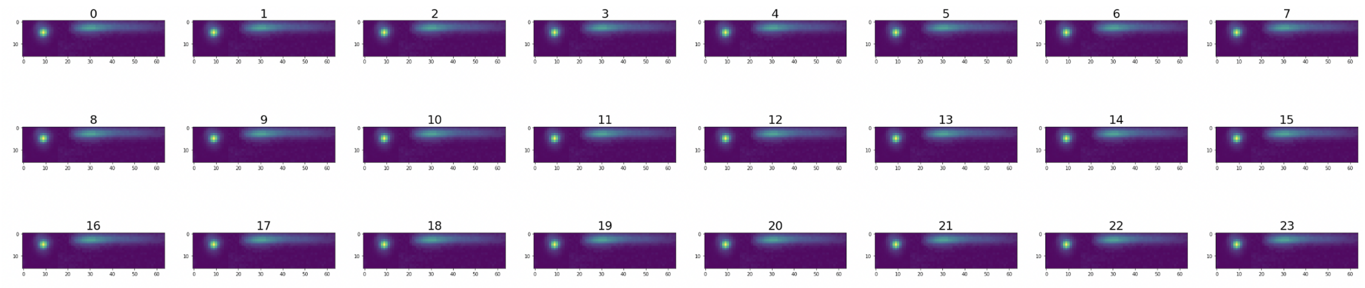

Figure 4: Corresponding high-SNR patches which were created by SF fusion (patch_sum__)

2 Training the model

The model training is performed by using second script. Thus first we need to import all the required libraries. This code shows how to upload the training set from *.npz achieves. Otherwise the training set can be concatenated from separately uploaded *.tif files (see the full code which is available on GitHub).

```
import keras
import tensorflow as tf
import os

import numpy as np
from matplotlib import pyplot as plt
import random
from tifffile import imread, imsave
from PIL import Image


from skimage import io, img_as_ubyte
from skimage.transform import resize, rescale
import random
import pandas as pd
from tqdm import tqdm

from tensorflow.keras.layers import Lambda,Input,Conv2D,BatchNormalization,AveragePooling2D,
                                    LeakyReLU,Conv2DTranspose,concatenate,UpSampling2D
                                    ,Dropout
from skimage.morphology import disk, white_tophat

from numpy import expand_dims
from keras.preprocessing.image import ImageDataGenerator
from keras.models import load_model
from sklearn.model_selection import train_test_split

image_width = 64
image_height = 16
image_chanels = 1

Xtest = np.load('.../data/Xtest.npz')
Xtest.files
```

```
X_test = Xtest['Xtest']

Xtrain = np.load('.../data/Xtrain.npz')
Xtrain.files
X_train = Xtrain['Xtrain']

Ytest = np.load('.../data/Ytest.npz')
Ytest.files
Y_test = Ytest['Ytest']

Ytrain = np.load('.../data/Ytrain.npz')
Ytrain.files
Y_train = Ytrain['Ytrain']
```

Now we can define the model

```

##############################################
#
#    /\    /\    ___   ___|  __  |
#   /  \  /  \  |   | |   | |__| |
#  /    \/    \ |___| |___| |__   |___
#
##############################################
inputs = Input((image_height, image_width, image_chanels))

c0=Conv2D(16, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(inputs)
c0=BatchNormalization(axis=-1)(c0)
c0=Conv2D(16, (6,6),activation = LeakyReLU(alpha=0.2),strides = 2,kernel_initializer='he_normal',
                                                    padding = 'same')(c0)
c0=BatchNormalization(axis=-1)(c0)

c1=Conv2D(32, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(c0)
c1=BatchNormalization(axis=-1)(c1)
c1=Conv2D(32, (6,6),activation = LeakyReLU(alpha=0.2),strides = 2, kernel_initializer='he_normal',
                                                    padding = 'same')(c1)
c1=BatchNormalization(axis=-1)(c1)

c2=Conv2D(64, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(c1)
c2=BatchNormalization(axis=-1)(c2)
c2=Conv2D(64, (6,6),activation = LeakyReLU(alpha=0.2),strides = 2,kernel_initializer='he_normal',
                                                    padding = 'same')(c2)
c2=BatchNormalization(axis=-1)(c2)

c3=Conv2D(128, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(c2)
c3=BatchNormalization(axis=-1)(c3)

u4=Conv2DTranspose(64, (6,6), padding ='same')(c3)
u4=concatenate([u4,c2])
c4=UpSampling2D(size=2)(u4)
c4=Conv2D(64, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(c4)
c4=BatchNormalization(axis=-1)(c4)

u5=Conv2DTranspose(32, (6,6), padding ='same')(c4)
u5=concatenate([u5,c1])
c5=UpSampling2D(size=2)(u5)
c5=Conv2D(32, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(c5)
c5=BatchNormalization(axis=-1)(c5)

u6=Conv2DTranspose(16, (6,6), padding ='same')(c5)
u6=concatenate([u6,c0])
c6=UpSampling2D(size=2)(u6)
c6=Conv2D(16, (6,6),activation = LeakyReLU(alpha=0.2),kernel_initializer='he_normal', padding = '
                                                    same')(c6)
c6=BatchNormalization(axis=-1)(c6)

outputs = Conv2D(1, (1,1), activation ='sigmoid')(c6)

model = tf.keras.Model(inputs = [inputs], outputs = [outputs])
```

```
model.compile(optimizer = 'adam', loss="mean_squared_error")
model.summary()
```

And train it

```

reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2,patience=4, min_lr=0
                                                 .0001)
earlyStop = keras.callbacks.EarlyStopping(patience=4, verbose=1, restore_best_weights=True)
callbacks_list = [earlyStop, reduce_lr]

history = model.fit(X_train,Y_train,validation_split=0.1, batch_size=80, epochs=20)

model.save("path")
```

Further, the trained model can be used to process data in test set or other patches