REPORT

Exercise 4: Image Inpainting

Part 1: Introduction

We consider the problem of image inpainting. This problem consists in finding missing pixels in an image. The location of missing pixels is known and stored as a mask image. There is no information about the pixel values under the mask.

Part 2: First network structure

Q1. Write the function *buildDecoder* that build the model

First, let's declare our image dimensions

```python
img_height = 512
img_width = 512
```

Now let's define our function

```python
def buildDecoder(inputShape,nc,display=False):
    # Input
    Input = layers.Input(shape=inputShape)

    # Block 1
    x = layers.Conv2D(filters=nc[0], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 2
    x = layers.UpSampling2D(interpolation="bilinear")(x)
    x = layers.Conv2D(filters=nc[1], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 3
    x = layers.UpSampling2D(interpolation="bilinear")(x)
    x = layers.Conv2D(filters=nc[2], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 4
    x = layers.UpSampling2D(interpolation="bilinear")(x)
    x = layers.Conv2D(filters=nc[3], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 5
    x = layers.UpSampling2D(interpolation="bilinear")(x)
    x = layers.Conv2D(filters=nc[4], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 6
    x = layers.UpSampling2D(interpolation="bilinear")(x)
    x = layers.Conv2D(filters=nc[5], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Final block
    x = layers.Conv2D(filters=3, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False", activation="tanh")(x)

    # Model
    model = tf.keras.Model(Input, x)

    if display:
        print(model.summary())

    return model
```

Q2. Define the optimizer

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
```

Q3. Write the *trainStep* function. Here, I also defined the *NoiseInit* function of TD3. Since my code has run successfully, I uncommented @tf.function()

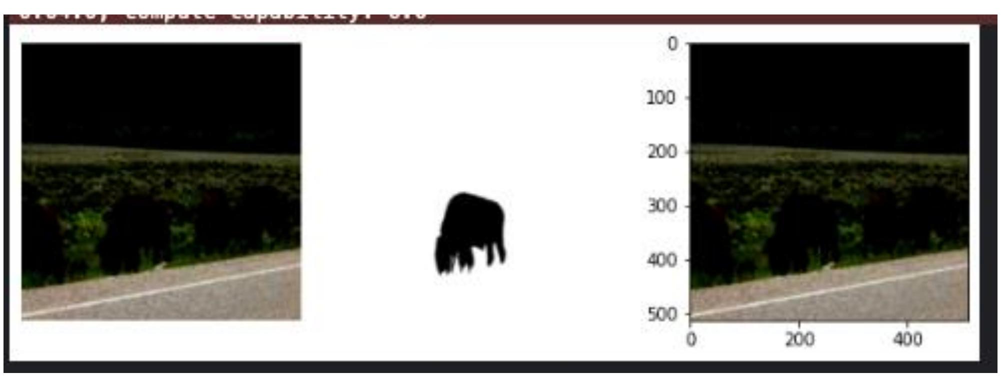
```
@tf.function()

def NoiseInit(height, width):
    Img=tf.random.uniform(shape=[height,width,3], minval=-0.1, maxval=0.1, dtype=tf.float32)
    Imgs=tf.expand_dims(Img,axis=0)
    return Imgs

def trainStep(imgInput,imgTarget,imgMask,stddev):
    noise = tf.random.normal(shape=[1]+inputShape, stddev=stddev)

    print('*************** Tracing *********************')
    with tf.GradientTape() as tape:
        imgResult = model(imgInput+noise, training=True)
#         print(imgInput.shape)
#         print(imgResult.shape)

        masked_error = tf.multiply((tf.math.add(imgTarget,-imgResult)), imgMask)
        loss = tf.reduce_mean(tf.square(masked_error))

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients,model.trainable_variables))

    return loss,imgResult
```

Q4. Write the inpainting optimization loop

```
def inpatining(nbiter,stddev):
    allLoss = []
    nsamples = 100

    for iter in tqdm(range(nbiter)):
        loss, imgResult = trainStep(imgInput,imgTarget,imgMask,stddev)
        allLoss.append(loss)

        if iter%100 == 0:
            display(imgTarget,imgResult,allLoss)

    return imgResult[0],allLoss
```

Q5. Display the input, mask and target image

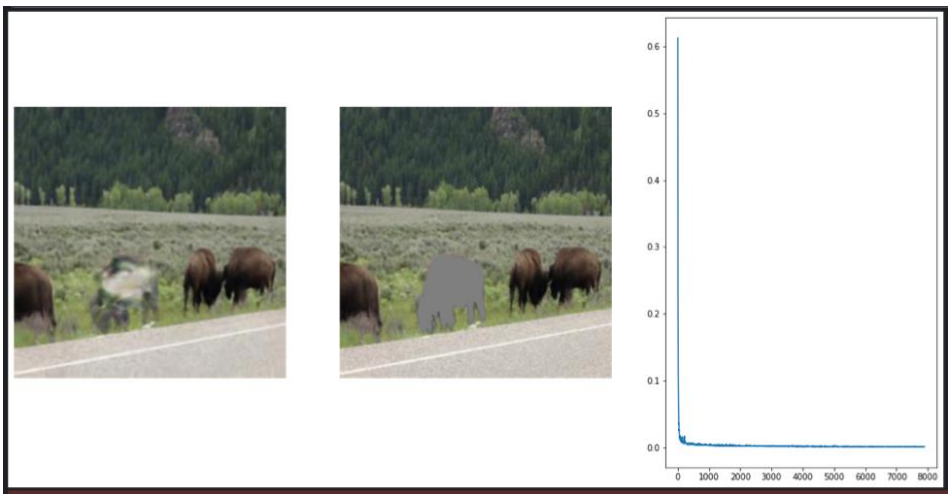
```python
no=4
if no==1:
    # building. Pick a mask buildingMask1, buildingMask2 or buildingMask3
    filenameImg=os.path.join(baseFolder,'building.png')
    filenameMask=os.path.join(baseFolder,'buildingMask1.png')
elif no==2:
    # abstract wall paper
    filenameImg=os.path.join(baseFolder,'0011_img.png')
    filenameMask=os.path.join(baseFolder,'0011_mask.png')
elif no==3:
    # dog
    filenameImg=os.path.join(baseFolder,'0071_img.png')
    filenameMask=os.path.join(baseFolder,'0071_mask.png')
elif no==4:
    # bison
    filenameImg=os.path.join(baseFolder,'0090_img.png')
    filenameMask=os.path.join(baseFolder,'0090_mask.png')
elif no==5:
    filenameImg=os.path.join(baseFolder,'0063_img.png')
    filenameMask=os.path.join(baseFolder,'0063_mask.png')
elif no==6:
    filenameImg=os.path.join(baseFolder,'0089_img.png')
    filenameMask=os.path.join(baseFolder,'0089_mask.png')

# read 2 images with read_image : original image + mask
imgOrigine = read_image(filenameImg,True)
imgMask = read_image(filenameMask,True,False)

# create masked image
imgTarget = tf.multiply(imgOrigine,imgMask)

# Display in a figure with subplot : the original image, the mask and the target
plt.figure(figsize=(10,15))
plt.subplot(1,3,1)
plt.imshow(imgOrigine[0])
plt.axis('off')
plt.subplot(1,3,2)
plt.imshow(imgMask[0])
plt.axis('off')
plt.subplot(1,3,3)
plt.imshow(imgTarget[0])
plt.show()
```



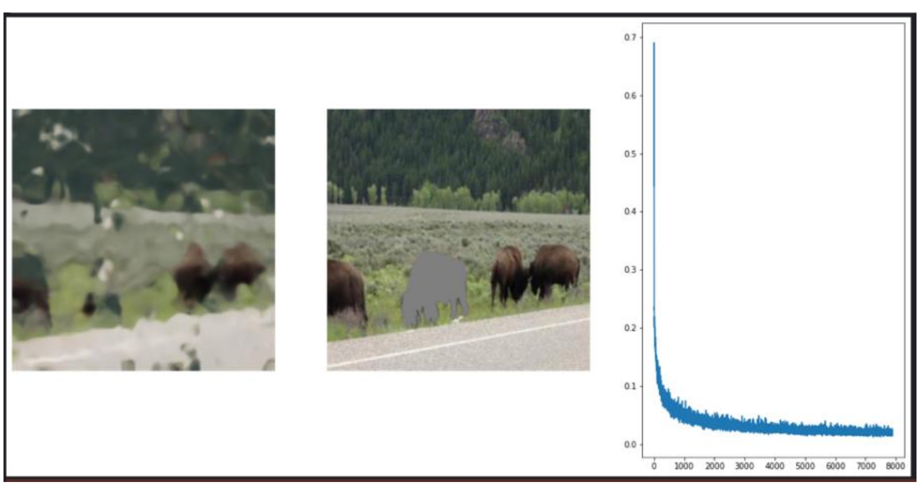
Q6. Main program

Training with a standard deviation of 0/100

```
stddev = 0/100 # 2/100
nbiter = 8000

nfilter = [128]*6
upsample_factor = 2**(len(nfilter)-1)
inputShape = [img_height//upsample_factor,img_width//upsample_factor,3]

imgInput = NoiseInit(inputShape[0], inputShape[1])

model = buildDecoder(inputShape,nfilter,display=False)

imgResult, allLoss = inpatining(nbiter,stddev)

# display final result
plt.figure(figsize=[15,10])
plt.imshow(imgResult)
plt.show()
```



Training with a standard deviation of 2/100



Q7. This approach won't perform well when the mask region is very large because we are up sampling the image. Since pixels are "guessed", it's better to use for small mask so the difference between produced image and the original image will not be big.

Part 3: Advanced networks

Q8. The role of 1x1 first convolutional layer with n1[0] channels is to increase the number of channels of the image without changing its actual dimension (width and height). Furthermore, the number of channels of this layer should be equal to the number of the next layer because the next block is a residual block. And in order to perform an identity in parallel to 3 blocks including in the residual block, we should have an equal number of channels.

Q9. The structure can only work for n1[i] = n1[i-1] for the same reason as mentioned in Q8. Imagine we have n1[i] different than n1[i-1]. As in a residual block, we will perform the sum of the output of 3 convolutional blocks and the identity. If the input of the residual block has a different number of channels than the output, then the identity will have a different number of channels. Consequently, the operation cannot be performed.

Q10. A colored image always has 3 or 4 channels. In our case, we use RGB so the number of channels should be 3.

Q11. Define a residual block

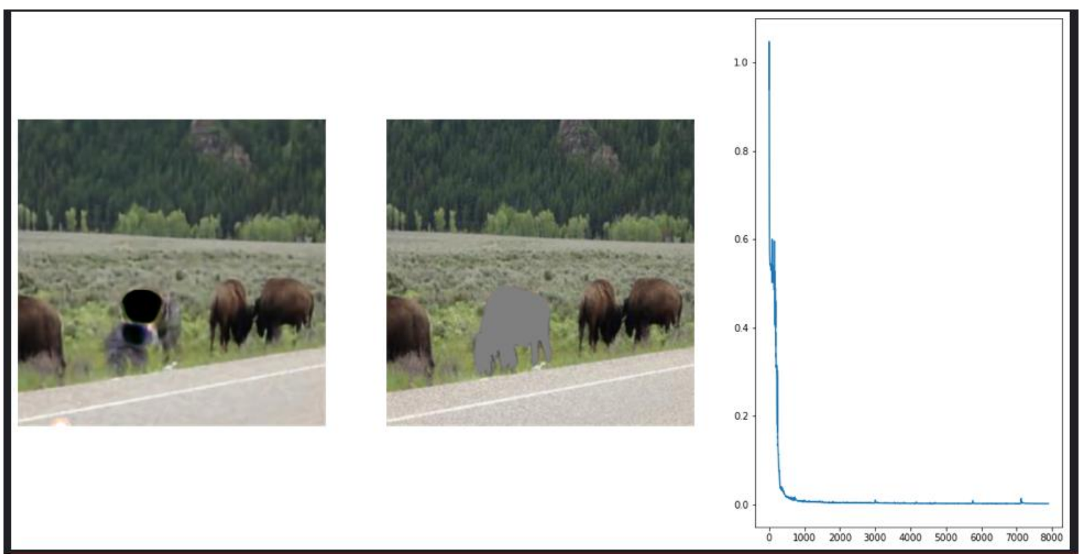
```python
def residual_block(Input, k, n1, n2):
    identity = tf.identity(Input)

    # Block 1
    Input = layers.Conv2D(filters=n1, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    Input = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(Input)
    Input = layers.ReLU()(Input)

    # Block 2
    Input = layers.Conv2D(filters=n2, kernel_size=k, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    Input = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(Input)
    Input = layers.ReLU()(Input)

    # Block 3
    Input = layers.Conv2D(filters=n1, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    Input = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(Input)
    Input = layers.ReLU()(Input)

    # Residual
    Input += identity

    return Input
```

Q12. Generate a simplified residual network

```python
def genereResidual2(inputShape, n1, n2, display=False):
    # Input
    Input = layers.Input(shape=inputShape)

    # Block 0: 1x1 convolution
    x = layers.Conv2D(filters=n1[0], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)

    # Block 1
    x = residual_block(x, 3, n1[0], n2[0])

    # From Block 2 until before Final Block
    for i in range(1, len(n1)):
        x = layers.UpSampling2D(interpolation="bilinear")(x)
        x = residual_block(x, 3, n1[i], n2[i])

    # Final block
    x = layers.Conv2D(filters=3, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False", activation="tanh")(x)

    # Model
    model = tf.keras.Model(Input, x)

    if display:
        print(model.summary())

    return model
```

Q13. Main program for our simplified residual network

```python
stddev = 0/100
nbiter = 8000

n1 = [128]*6
n2 = [64]*6
upsample_factor = 2**(len(n1)-1)
inputShape = [img_height//upsample_factor,img_width//upsample_factor,3]

imgInput = NoiseInit(inputShape[0], inputShape[1])

model = genereResidual2(inputShape, n1, n2, display=False)

imgResult, allLoss = inpatining(nbiter,stddev)

# display final result
plt.figure(figsize=[15,10])
plt.imshow(imgResult)
plt.show()
```



Q14. “Advanced residual network”

Define a function that generate an advanced residual block

```python
def residual_block_advanced(Input, k, n1, n2):

    # Additional Branch
    additional_branch = tf.identity(Input)
    additional_branch = layers.Conv2D(filters=n1, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(additional_branch)
    additional_branch = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(additional_branch)
    additional_branch = layers.ReLU()(additional_branch)

    # Block 1
    Input = layers.Conv2D(filters=n1, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    Input = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(Input)
    Input = layers.ReLU()(Input)

    # Block 2
    Input = layers.Conv2D(filters=n2, kernel_size=k, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    Input = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(Input)
    Input = layers.ReLU()(Input)

    # Block 3
    Input = layers.Conv2D(filters=n1, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)
    Input = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(Input)
    Input = layers.ReLU()(Input)

    # Residual
    Input += additional_branch

    return Input
```

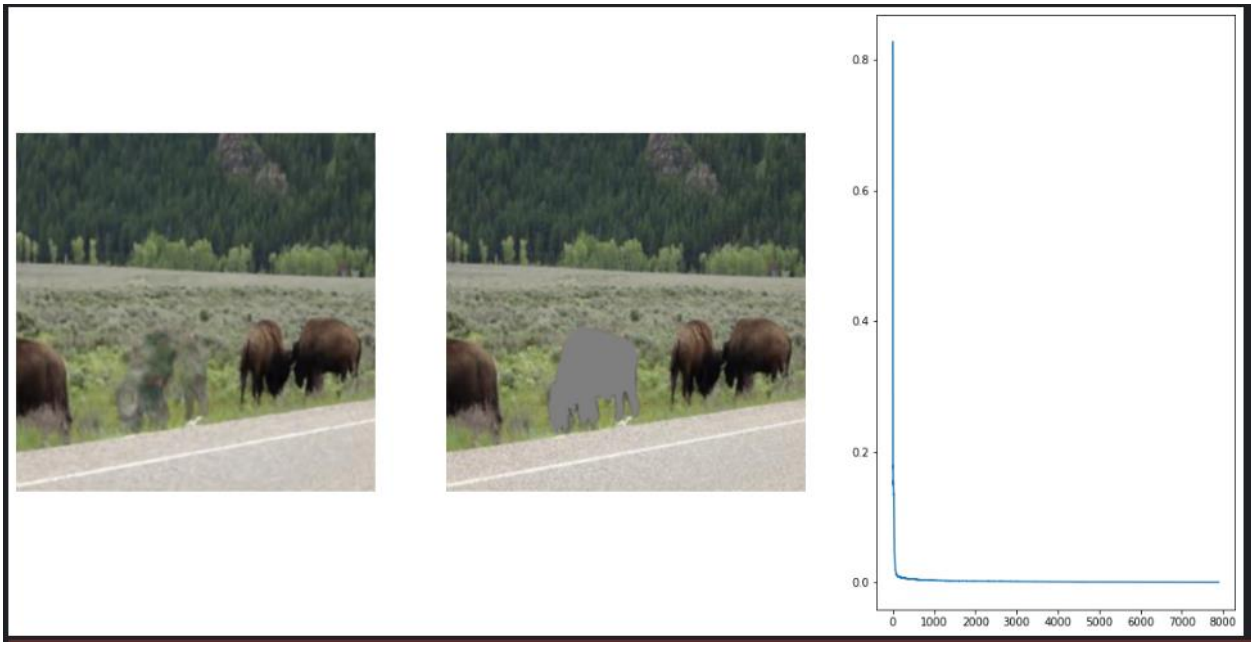
Generate an advanced residual network

```python
def genereAdvancedResidual2(inputShape, n1, n2, display=False):
    # Input
    Input = layers.Input(shape=inputShape)

    # Block 0: 1x1 convolution
    x = layers.Conv2D(filters=n1[0], kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(Input)

    # Block 1
    x = residual_block_advanced(x, 3, n1[0], n2[0])

    # From Block 2 until before Final Block
    for i in range(1, len(n1)):
        x = layers.UpSampling2D(interpolation="bilinear")(x)
        x = residual_block_advanced(x, 3, n1[i], n2[i])

    # Final block
    x = layers.Conv2D(filters=3, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False", activation="tanh")(x)

    # Model
    model = tf.keras.Model(Input, x)

    if display:
        print(model.summary())

    return model
```

Main program

```python
stddev = 1/100
nbiter = 8000

n1 = [256//2**i for i in range(5)]
n2 = [128//2**i for i in range(5)]
upsample_factor = 2**(len(n1)-1)
inputShape = [img_height//upsample_factor,img_width//upsample_factor,3]

imgInput = NoiseInit(inputShape[0], inputShape[1])

model = genereAdvancedResidual2(inputShape, n1, n2, display=False)

imgResult, allLoss = inpatining(nbiter,stddev)

# display final result
plt.figure(figsize=[15,10])
plt.imshow(imgResult)
plt.show()
```

Result



Q15. UNet Architecture

Define a function that generate down-conv block, up-conv block and skip block

```
def down_conv_block(x, nd, kd):

    # Block 1
    x = layers.Conv2D(filters=nd, kernel_size=kd, kernel_initializer=tf.keras.initializers.HeNormal(), strides=2, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 2
    x = layers.Conv2D(filters=nd, kernel_size=kd, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    return x

def up_conv_block(x, nu, ku):

    # Block 1
    x = layers.Conv2D(filters=nu, kernel_size=ku, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 2
    x = layers.Conv2D(filters=nu, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    # Block 3
#     if first_block:
    x = layers.UpSampling2D(interpolation="bilinear")(x)

    return x

def skip_block(x, ns):

    # Block 1
    x = layers.Conv2D(filters=ns, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x)
    x = layers.ReLU()(x)

    return x
```

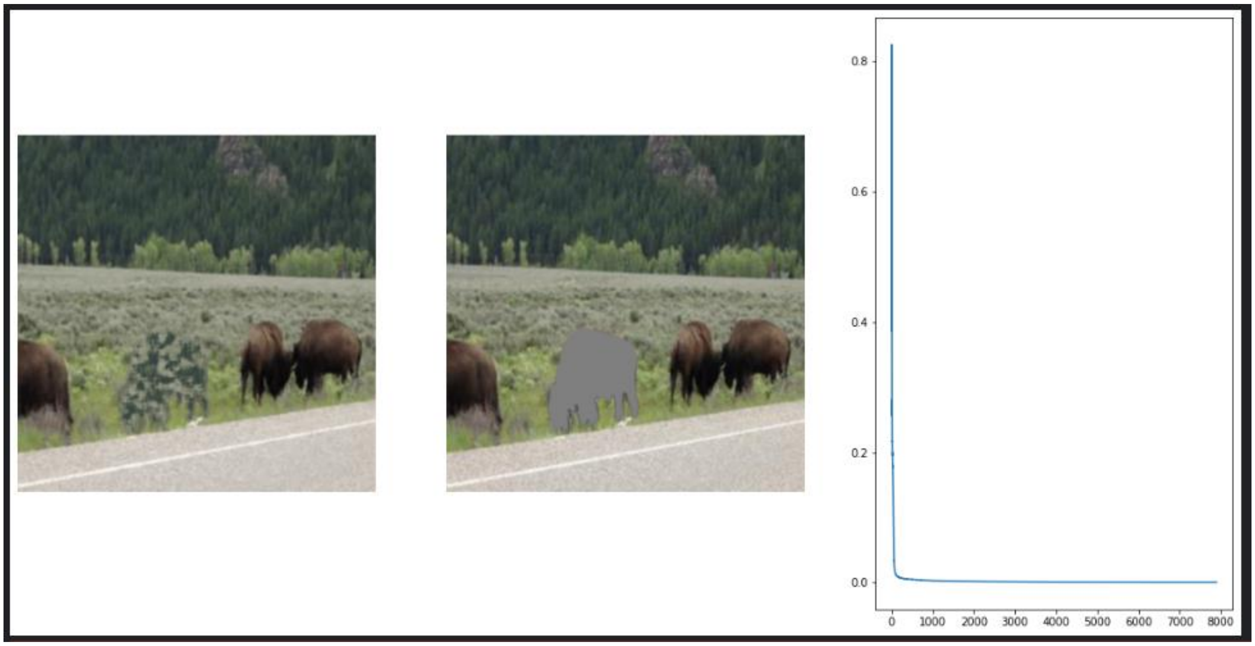
Generate an UNet network

```python
def genereUNet(inputShape, nd, kd, nu, ku, ns, display=False):
    # Input
    Input = layers.Input(shape=inputShape)

    # Output lists:
    output_down = []
    output_skip = []

    x = down_conv_block(Input, nd[0], kd[0])
    output_down.append(x)

    for i in range(1,len(nd)):
        x = down_conv_block(x, nd[i], kd[i])
        output_down.append(x)

    for i in range(len(ns)):
        x = skip_block(output_down[i], ns[i])
        output_skip.append(x)

    x = up_conv_block(output_skip[4], nu[4], ku[4])

    for i in reversed(range(1,len(nu)-1)):
        x = up_conv_block(output_skip[i], nu[i], ku[i])
        x = tf.keras.layers.Concatenate()([x, output_skip[i-1]])

    x = up_conv_block(output_skip[0], nu[0], ku[0])

    # Final block
    x = layers.Conv2D(filters=3, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False", activation="tanh")(x)

    # Model
    model = tf.keras.Model(Input, x)

    if display:
        print(model.summary())

    return model, output_skip
```

Main program

```python
stddev = 1/100
nbiter = 8000

nu = [128]*5
nd = [128]*5
ns = [4]*5
kd = [3]*5
ku = [5]*5

inputShape = [img_height, img_width, 3]

imgInput = NoiseInit(inputShape[0], inputShape[1])

model, output_skip = genereUNet(inputShape, nd, kd, nu, ku, ns, display=False)

imgResult, allLoss = inpatining(nbiter,stddev)

# display final result
plt.figure(figsize=[15,10])
plt.imshow(imgResult)
plt.show()
```

Result



Q16. Inception

Function that generates one inception block

```python
def inception_block(x, n1, n3, n5, nd, np):

    # Branch 1
    x1 = layers.Conv2D(filters=n1, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x1 = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x1)
    x1 = layers.ReLU()(x1)

    # Branch 2 - Block 1
    x2 = layers.Conv2D(filters=nd, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x2 = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x2)
    x2 = layers.ReLU()(x2)

    # Branch 2 - Block 2
    x2 = layers.Conv2D(filters=n3, kernel_size=3, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x2)
    x2 = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x2)
    x2 = layers.ReLU()(x2)

    # Branch 3 - Block 1
    x3 = layers.Conv2D(filters=nd, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x)
    x3 = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x3)
    x3 = layers.ReLU()(x3)

    # Branch 3 - Block 2
    x3 = layers.Conv2D(filters=n5, kernel_size=3, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x3)

    # Branch 3 - Block 3
    x3 = layers.Conv2D(filters=n5, kernel_size=3, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x3)
    x3 = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x3)
    x3 = layers.ReLU()(x3)

    # Branch 4 - Block 1
    x4 = layers.MaxPool2D(strides=1, padding="same")(x)

    # Branch 4 - Block 2
    x4 = layers.Conv2D(filters=np, kernel_size=1, kernel_initializer=tf.keras.initializers.HeNormal(), strides=1, padding="same", use_bias="False")(x4)
    x4 = layers.BatchNormalization(momentum=0.9, epsilon=1e-5)(x4)
    x4 = layers.ReLU()(x4)

    # Concatenation block
    x = tf.keras.layers.Concatenate()([x1, x2, x3, x4])

    return x
```

I stop here because the subject doesn't mention any information about the general architecture of Inception and no information about n1, n3, n5, nd and np