Image Inpainting

Course Project CS 736

Submitted by :

Neeraj Nixon (20D070056)

Ananthapadmanabhan A(20d070010)

What is image inpainting

- Refers to the technique of filling in parts of a damaged image so that it looks undamaged.
- Can be done manually but usually done using specific algorithms.
- It usually requires a good understanding of the context and content of the image to recreate it with any level of naturality
- Difficult to do on general image data but having specific context on the image makes it easier to train models for inpainting.
- Usually used for recreating historical pictures, missing data from scans, filling in blind spots of equipment etc.

Why use Inpainting

- Almost all scan data from Medical equipment have some form of noise or artifacts in the raw data.
- Patches with high detail might be corrupted by noise so that a wrong diagnosis may happen.
- Image inpainting can be used as a method to accurately determine the missing/corrupted patches of the image which might lead to a better diagnosis.
- Much easier to do inpainting for medical equipment as we have a lot of image specific information which can be used to train the model for even better results.

Inpainting methods

There are many Inpainting methods which work pretty well in specific scenarios.

- Patch-based methods: This approach involves finding similar patches in the surrounding area of the missing region and using them to fill in the gap. These methods work well when the missing areas are small or when the image has repetitive patterns.
- Deep learning-based methods: These methods use convolutional neural networks (CNNs) to learn the underlying structure and texture of an image. They have shown promising results in various image inpainting tasks, including medical imaging.
- Partial differential equation (PDE)-based methods: These methods model the impainting process as a diffusion process, where the missing pixels are gradually filled in by propagating information from the surrounding regions. They work well for smooth and continuous images.
- Exemplar-based methods: These methods use a database of similar images to the input image to generate the inpainted result. They are effective in handling complex textures and structures in the missing regions of context specific images.
- Hybrid methods: These methods combine two or more of the above techniques to leverage their respective strengths and overcome their limitations.

Using Deep Learning for Inpainting

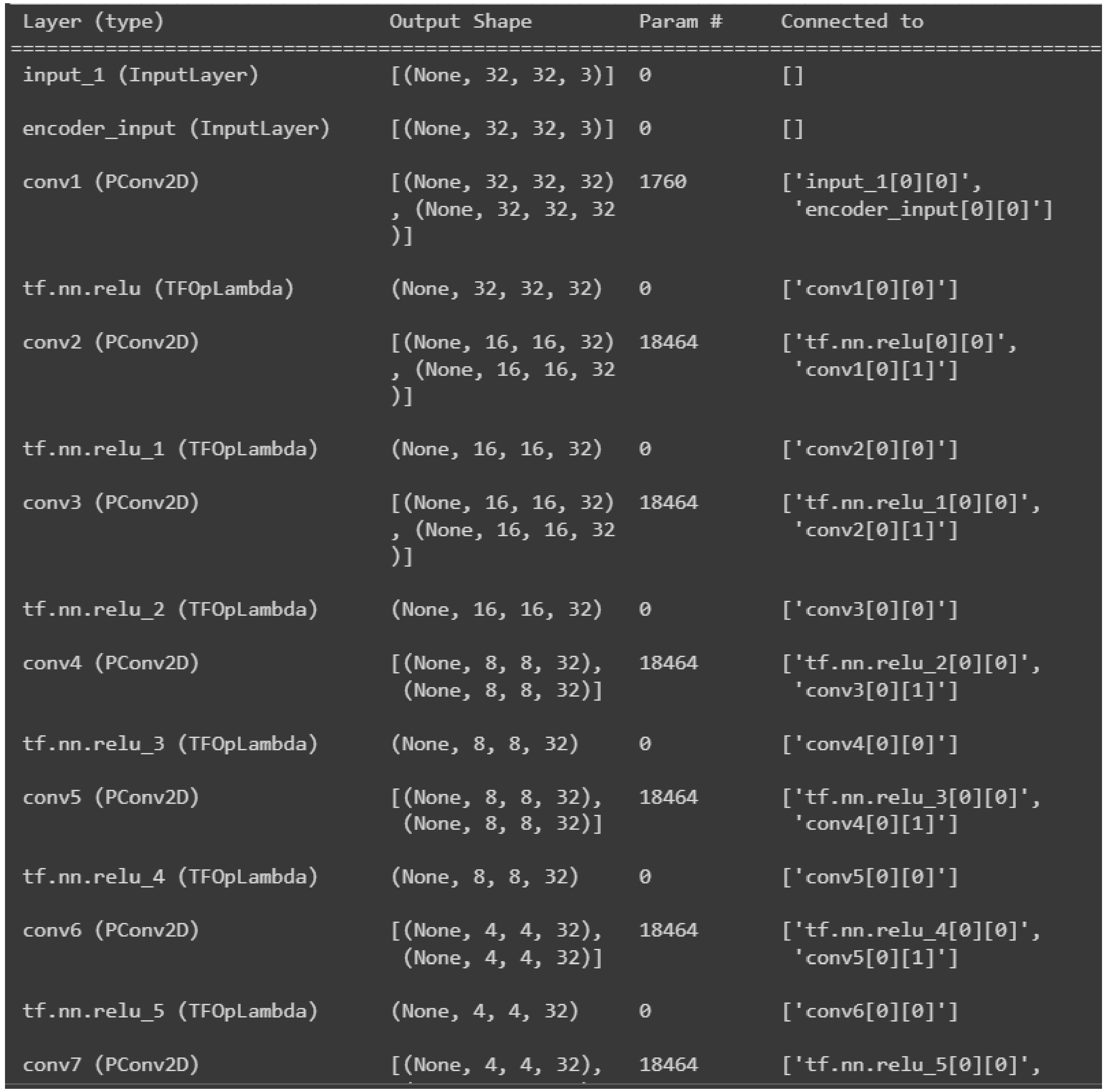
Deep Learning in image Inpainting has shown a lot of promise in processing of medical images even when there are large data chunks missing.

- Context Encoder (CE): CE is a deep neural network-based method that uses a convolutional neural network (CNN) to learn the underlying structure of the image and generate the missing regions.
- GAN-based methods have been used for image inpainting by training the generator to fill in the missing regions while the discriminator tries to differentiate between the generated images and the real images. The generator is then trained to generate images that fool the discriminator into thinking they are real.
- Deep Exemplar-based Image Inpainting (DEI): DEI is a deep learning-based method that uses a combination of exemplar-based and CNN-based inpainting. It first selects similar patches from the available image data and then uses a CNN to generate the inpainted result.

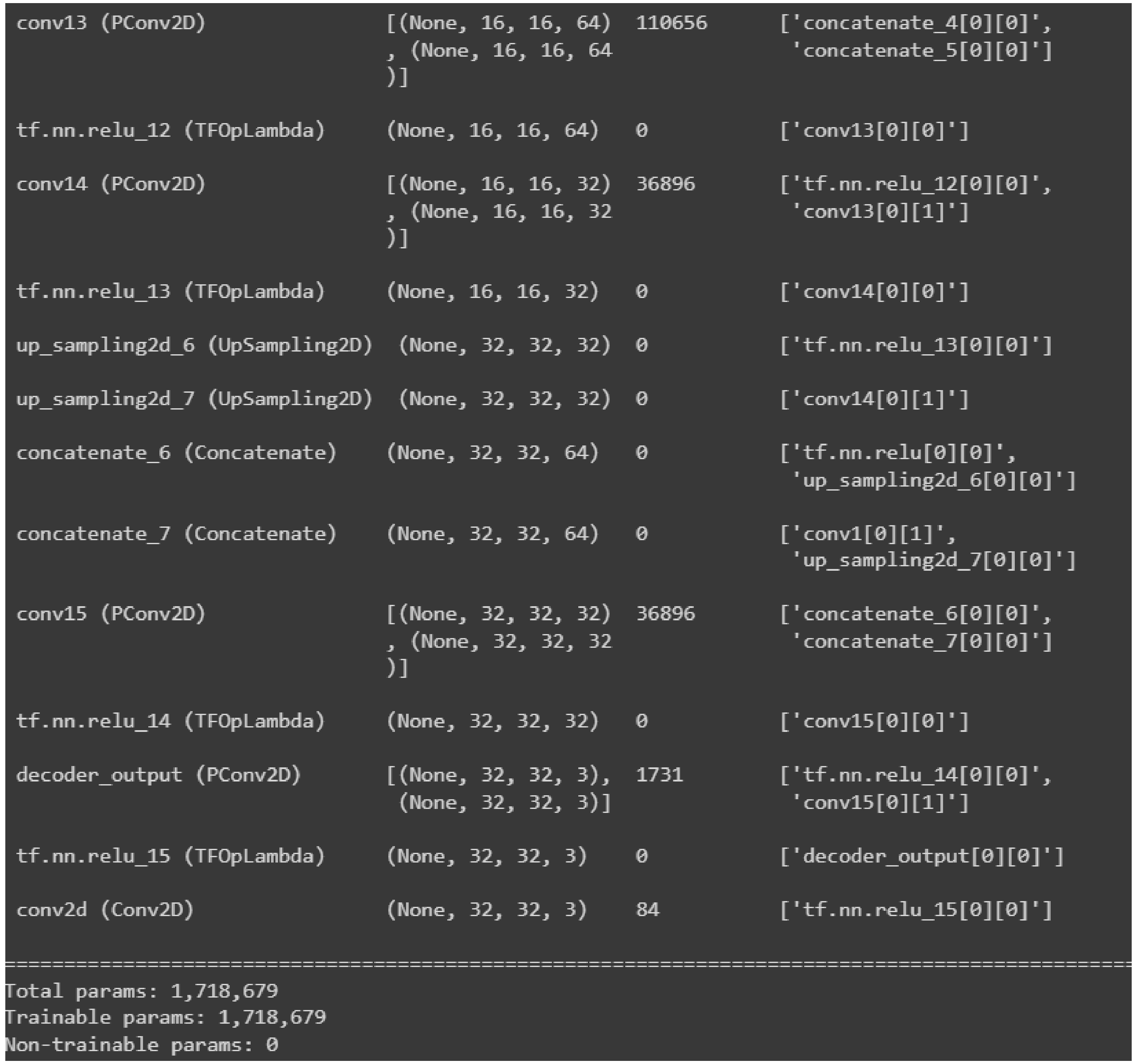
Methods

- We have implemented two methods of image inpainting in our project and tested the results of the models on the CIFA10 dataset. We use low resolution images so as to cut down on computation time since inpainting can be done on any image resolution
- The method implemented is a variation of the context encoder method. Here we use auto encoders to compress the data and then decoding the compressed data to get the inpainted image.
- We have used Peak Signal to Noise Ratio(PSNR) as the loss metric for comparison of test results.
- We have also used different auto encoders, i.e UNet like vanilla CNN and partial CNN's to compare the better metric for inpainting.

UNet like architecture for vanilla autoencoder



```
 Layer (type)                 Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)         [(None, 32, 32, 3)]  0           []

 encoder_input (InputLayer)   [(None, 32, 32, 3)]  0           []

 conv1 (PConv2D)              [(None, 32, 32, 32)  1760        ['input_1[0][0]',
                              , (None, 32, 32, 32               'encoder_input[0][0]']
                              )]

 tf.nn.relu (TFOpLambda)      (None, 32, 32, 32)   0           ['conv1[0][0]']

 conv2 (PConv2D)              [(None, 16, 16, 32)  18464       ['tf.nn.relu[0][0]',
                              , (None, 16, 16, 32               'conv1[0][1]']
                              )]

 tf.nn.relu_1 (TFOpLambda)    (None, 16, 16, 32)   0           ['conv2[0][0]']

 conv3 (PConv2D)              [(None, 16, 16, 32)  18464       ['tf.nn.relu_1[0][0]',
                              , (None, 16, 16, 32               'conv2[0][1]']
                              )]

 tf.nn.relu_2 (TFOpLambda)    (None, 16, 16, 32)   0           ['conv3[0][0]']

 conv4 (PConv2D)              [(None, 8, 8, 32),   18464       ['tf.nn.relu_2[0][0]',
                               (None, 8, 8, 32)]                'conv3[0][1]']

 tf.nn.relu_3 (TFOpLambda)    (None, 8, 8, 32)     0           ['conv4[0][0]']

 conv5 (PConv2D)              [(None, 8, 8, 32),   18464       ['tf.nn.relu_3[0][0]',
                               (None, 8, 8, 32)]                'conv4[0][1]']

 tf.nn.relu_4 (TFOpLambda)    (None, 8, 8, 32)     0           ['conv5[0][0]']

 conv6 (PConv2D)              [(None, 4, 4, 32),   18464       ['tf.nn.relu_4[0][0]',
                               (None, 4, 4, 32)]                'conv5[0][1]']

 tf.nn.relu_5 (TFOpLambda)    (None, 4, 4, 32)     0           ['conv6[0][0]']

 conv7 (PConv2D)              [(None, 4, 4, 32),   18464       ['tf.nn.relu_5[0][0]',
```



```
 conv13 (PConv2D)                [(None, 16, 16, 64)  110656      ['concatenate_4[0][0]',
                                 , (None, 16, 16, 64               'concatenate_5[0][0]']
                                 )]

 tf.nn.relu_12 (TFOpLambda)      (None, 16, 16, 64)   0           ['conv13[0][0]']

 conv14 (PConv2D)                [(None, 16, 16, 32)  36896       ['tf.nn.relu_12[0][0]',
                                 , (None, 16, 16, 32               'conv13[0][1]']
                                 )]

 tf.nn.relu_13 (TFOpLambda)      (None, 16, 16, 32)   0           ['conv14[0][0]']

 up_sampling2d_6 (UpSampling2D)  (None, 32, 32, 32)   0           ['tf.nn.relu_13[0][0]']

 up_sampling2d_7 (UpSampling2D)  (None, 32, 32, 32)   0           ['conv14[0][1]']

 concatenate_6 (Concatenate)     (None, 32, 32, 64)   0           ['tf.nn.relu[0][0]',
                                                                   'up_sampling2d_6[0][0]']

 concatenate_7 (Concatenate)     (None, 32, 32, 64)   0           ['conv1[0][1]',
                                                                   'up_sampling2d_7[0][0]']

 conv15 (PConv2D)                [(None, 32, 32, 32)  36896       ['concatenate_6[0][0]',
                                 , (None, 32, 32, 32               'concatenate_7[0][0]']
                                 )]

 tf.nn.relu_14 (TFOpLambda)      (None, 32, 32, 32)   0           ['conv15[0][0]']

 decoder_output (PConv2D)        [(None, 32, 32, 3),  1731        ['tf.nn.relu_14[0][0]',
                                  (None, 32, 32, 3)]               'conv15[0][1]']

 tf.nn.relu_15 (TFOpLambda)      (None, 32, 32, 3)    0           ['decoder_output[0][0]']

 conv2d (Conv2D)                 (None, 32, 32, 3)    84          ['tf.nn.relu_15[0][0]']

==================================================================================================
Total params: 1,718,679
Trainable params: 1,718,679
Non-trainable params: 0
```

Partial CNN

- We replace normal convolution with partial convolution for this model which can be mathematically represented as:

$$x' = \begin{cases} \mathbf{W}^T(\mathbf{X} \odot \mathbf{M}) \frac{\text{sum}(\mathbf{1})}{\text{sum}(\mathbf{M})} + b, & \text{if } \text{sum}(\mathbf{M}) > 0 \\ 0, & \text{otherwise} \end{cases}$$

- X is the feature values for the current sliding (convolution) window, and M is the corresponding binary mask. Let the holes be denoted by 0 and non-holes by 1
- The scaling factor, sum(1)/sum(M), applies appropriate scaling to adjust for the varying amount of valid (unmasked) inputs.
- After each partial convolution operation, we update our mask as follows: if the convolution was able to condition its output on at least one valid input (feature) value, then we mark that location to be valid. It can be expressed as,

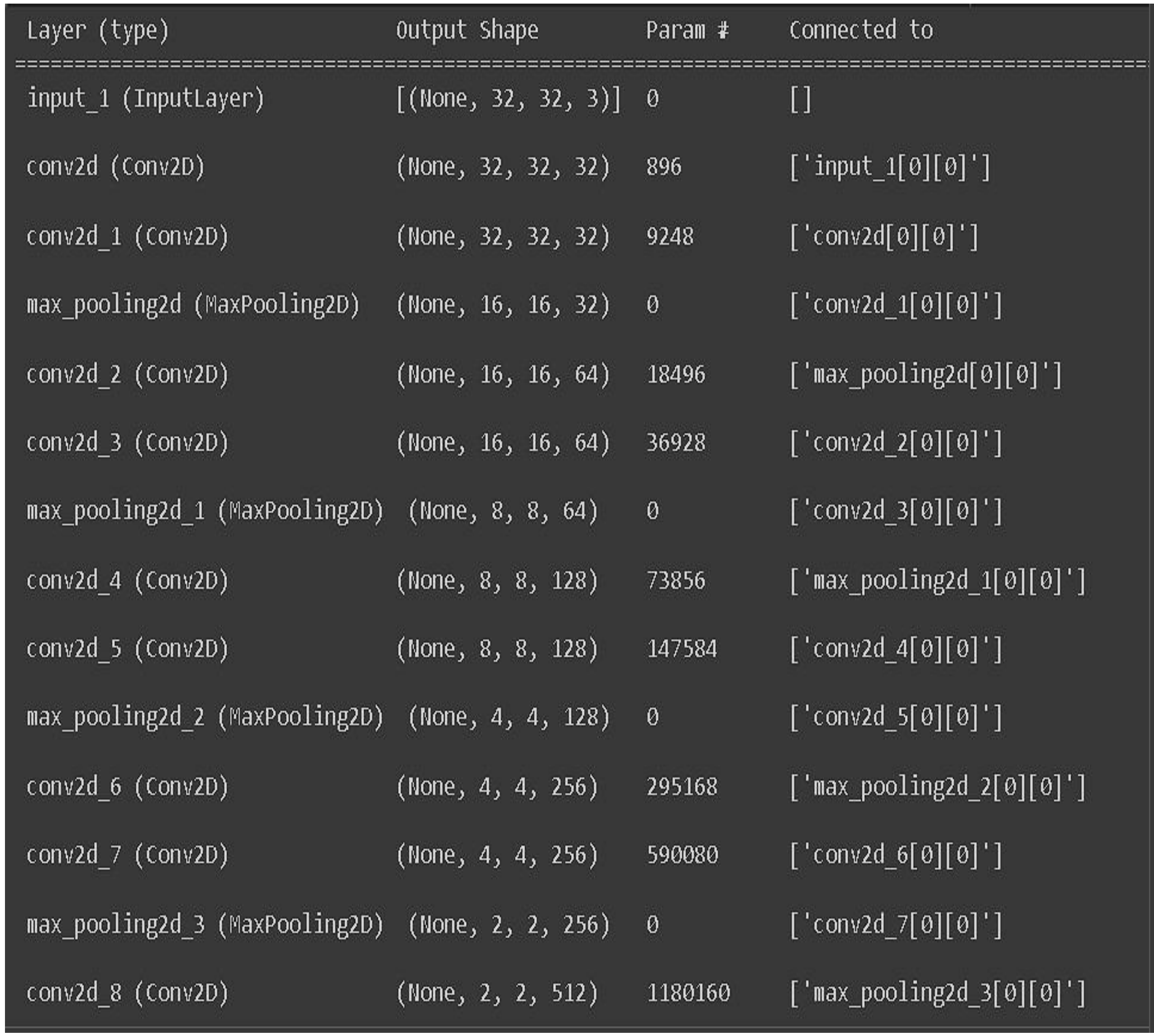
$$m' = \begin{cases} 1, & \text{if } \text{sum}(\mathbf{M}) > 0 \\ 0, & \text{otherwise} \end{cases}$$

- With multiple layers of partial convolutions, any mask will eventually be all ones, if the input contained any valid pixels.

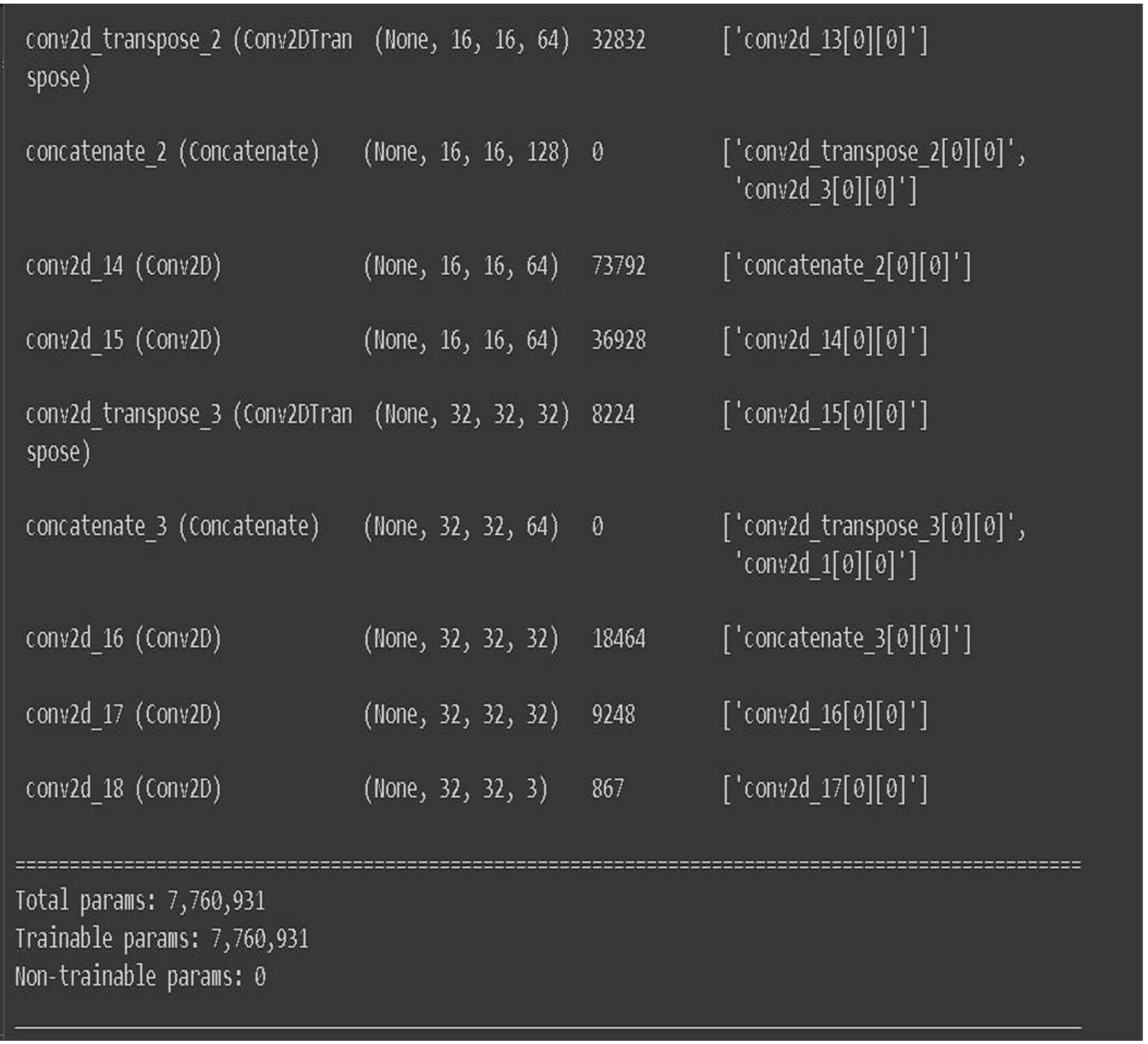
Why use partial CNN

- Normal CNN uses a sliding window over the whole picture even the damaged parts.
- This might be bad for the training model and produce wrong results especially if large parts of the image are damaged.
- Partial CNN used a mask to determine the damaged parts of the image and considers only the good parts of the image while training the data.
- Therefore Partial CNN should give us better image inpainting results with respect to Normal CNN’s

Partial convolutional autoencoder architecture

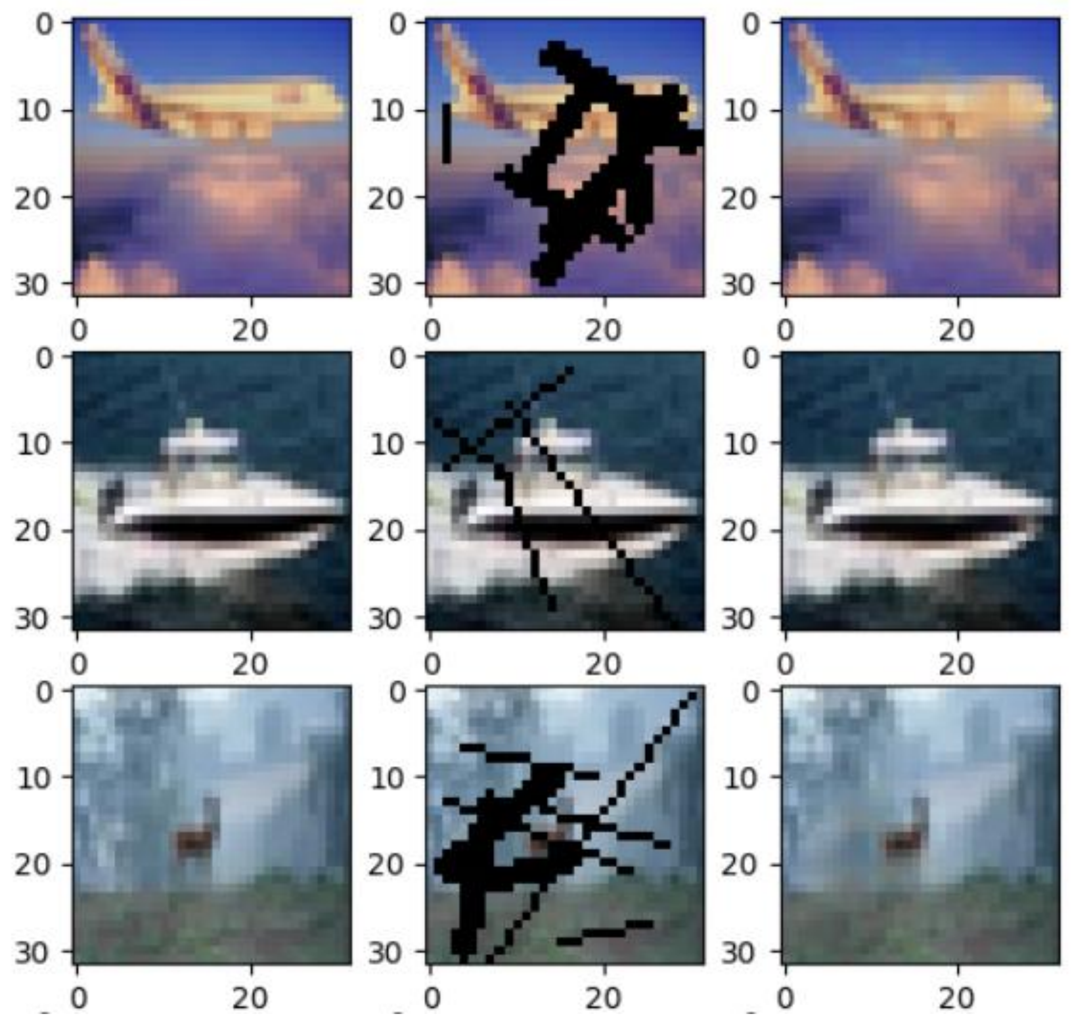


```
Layer (type)                  Output Shape         Param #     Connected to
==================================================================================
input_1 (InputLayer)          [(None, 32, 32, 3)]  0           []

conv2d (Conv2D)               (None, 32, 32, 32)   896         ['input_1[0][0]']

conv2d_1 (Conv2D)             (None, 32, 32, 32)   9248        ['conv2d[0][0]']

max_pooling2d (MaxPooling2D)  (None, 16, 16, 32)   0           ['conv2d_1[0][0]']

conv2d_2 (Conv2D)             (None, 16, 16, 64)   18496       ['max_pooling2d[0][0]']

conv2d_3 (Conv2D)             (None, 16, 16, 64)   36928       ['conv2d_2[0][0]']

max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 64)   0           ['conv2d_3[0][0]']

conv2d_4 (Conv2D)             (None, 8, 8, 128)    73856       ['max_pooling2d_1[0][0]']

conv2d_5 (Conv2D)             (None, 8, 8, 128)    147584      ['conv2d_4[0][0]']

max_pooling2d_2 (MaxPooling2D)  (None, 4, 4, 128)  0           ['conv2d_5[0][0]']

conv2d_6 (Conv2D)             (None, 4, 4, 256)    295168      ['max_pooling2d_2[0][0]']

conv2d_7 (Conv2D)             (None, 4, 4, 256)    590080      ['conv2d_6[0][0]']

max_pooling2d_3 (MaxPooling2D)  (None, 2, 2, 256)  0           ['conv2d_7[0][0]']

conv2d_8 (Conv2D)             (None, 2, 2, 512)    1180160     ['max_pooling2d_3[0][0]']
```



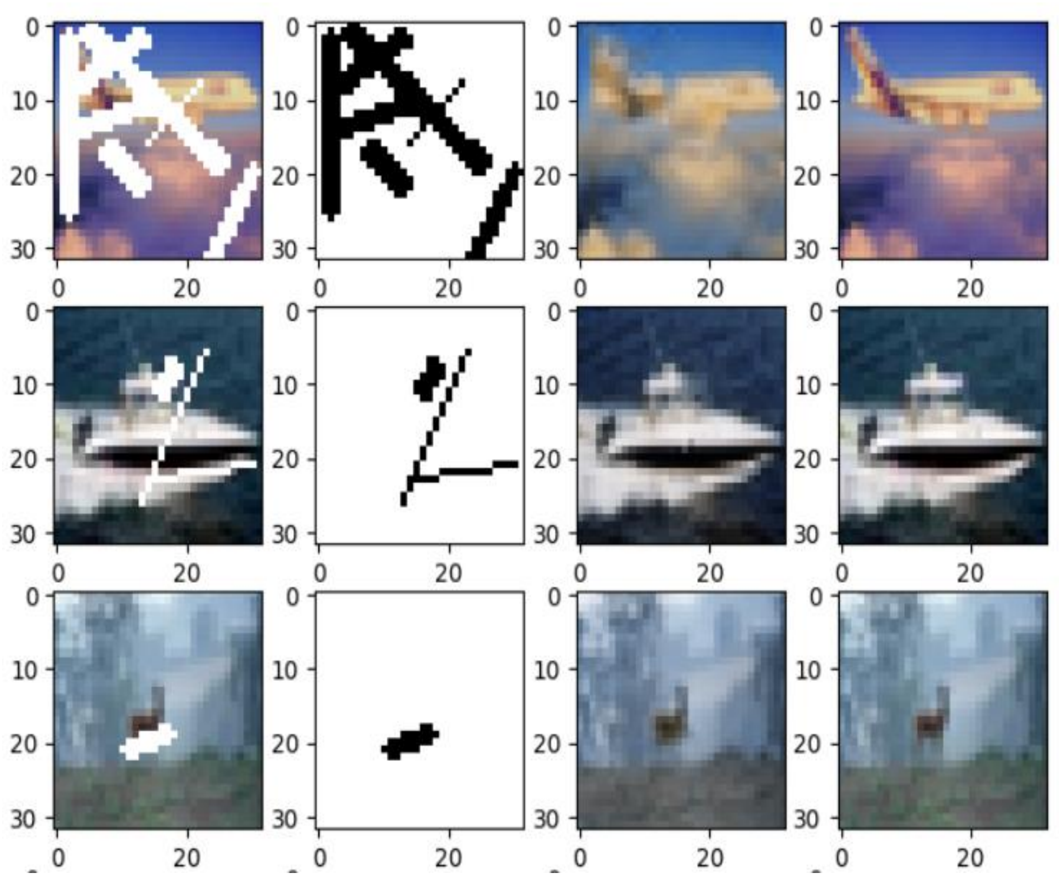
```
conv2d_transpose_2 (Conv2DTran  (None, 16, 16, 64)  32832      ['conv2d_13[0][0]']
spose)

concatenate_2 (Concatenate)    (None, 16, 16, 128)  0          ['conv2d_transpose_2[0][0]',
                                                                 'conv2d_3[0][0]']

conv2d_14 (Conv2D)             (None, 16, 16, 64)   73792      ['concatenate_2[0][0]']

conv2d_15 (Conv2D)             (None, 16, 16, 64)   36928      ['conv2d_14[0][0]']

conv2d_transpose_3 (Conv2DTran  (None, 32, 32, 32)  8224       ['conv2d_15[0][0]']
spose)

concatenate_3 (Concatenate)    (None, 32, 32, 64)   0          ['conv2d_transpose_3[0][0]',
                                                                 'conv2d_1[0][0]']

conv2d_16 (Conv2D)             (None, 32, 32, 32)   18464      ['concatenate_3[0][0]']

conv2d_17 (Conv2D)             (None, 32, 32, 32)   9248       ['conv2d_16[0][0]']

conv2d_18 (Conv2D)             (None, 32, 32, 3)    867        ['conv2d_17[0][0]']

==================================================================================
Total params: 7,760,931
Trainable params: 7,760,931
Non-trainable params: 0
__________________________________________________________________________________
```

Results-Vanilla CNN



- The first image is the original image, second image is the damaged image and the third image is the inpainted image.
- After 20 epochs, the validation loss was 0.0116 and PSNR was 78.56

Results- Partial CNN



- The first image is the masked image, second image is the binary mask, the third image is the inpainted image and fourth is the original image.
- After 20 epochs, the validation loss was 0.0276 and PSNR was 75.26

Inference & Conclusion

- From the above results we will notice that the vanilla CNN based image inpainting was slightly better than partial CNN image inpainting.
- This is mainly because of the fact that partial CNN is a complex architecture designed to work on high resolution images greater than 256 x 256 pixels.
- Although CNN based methods can give a decent enough inpainting, it creates boundary artifacts, distorted and blurry patches which requires post processing which is computationally expensive.

Inference & Conclusion

- We have only implemented loss function for pixel wise comparison which may make the model rigid. We can increase the efficiency of the model if we can use a multi pixel loss function.
- It should also be noted that we have used the PSNR as our loss metric. There are other loss functions which may increase the efficiency of CNN model such as SSIM.
- Also instead of CNN architectures, using generative adversarial networks and its variations may be much more effective for image inpainting.