FCN Patched Approach

Short Summary:

I have implemented a Fully Convolutional Neural Network (FCNN) for the given problem statement. In order to efficiently load and process the given data images in the FCNN, I have utilized a patch-based approach. The proposed solution mitigated the GPU usage and used less RAM as well. The proposed solution was able to achieve upto 98.78% accuracy in predicting the class labels for the test set.

Data Loading and Analysis:

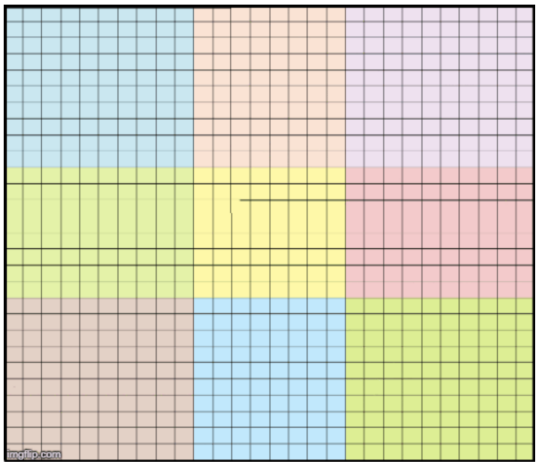
The given dataset consisted of a total of 200 JPEG images with 8 bottles in each frame, where each bottle corresponds to a class to be predicted, and 200 EXR images which were the masked outputs of the given JPEG images.

Fig 1: JPEG images corresponding to their masked EXR images.

The images in the dataset were resized to 540 x 640 and then stored into files by converting them into a byte stream by pickling the data using the pickle module. The pickle module helps in serializing and deserializing the data objects and is used as it allows the model to be saved in little time. It is also capable of keeping track of the objects it had previously serialized, thus it does not serialize the repeated references to the same object, and thus saves execution time.

Patch-based Approach:

Since the images in the dataset were of a significant size, I implemented a patch-based approach to efficiently store and process the images. First, the dataset is divided into training and test sets, with 80-20% ratio for training and testing. The images in the divided datasets are then broken into 26 patches of equal size (224x224) each. These patched images are used for training the model, and testing as well.



This is an example of how patching of an image works. The image is divided into patches of equal size and then the model is trained over these patches.

The patches are divided such that there is no miscalculation on the receptive field of the overlapping patches. This is called addressing boundary effects.

Fig 2: Example of patched image.

Architecture of FCNN used:

The FCNN architecture consists of two parts, namely, FCN encoder and FCN decoder. The architecture diagram is given below. Also, the model summary is given below the architecture diagram. The patches made out of the data images are passed into the FCNN model and the obtained images are then reconstructed to obtain the final outputs.

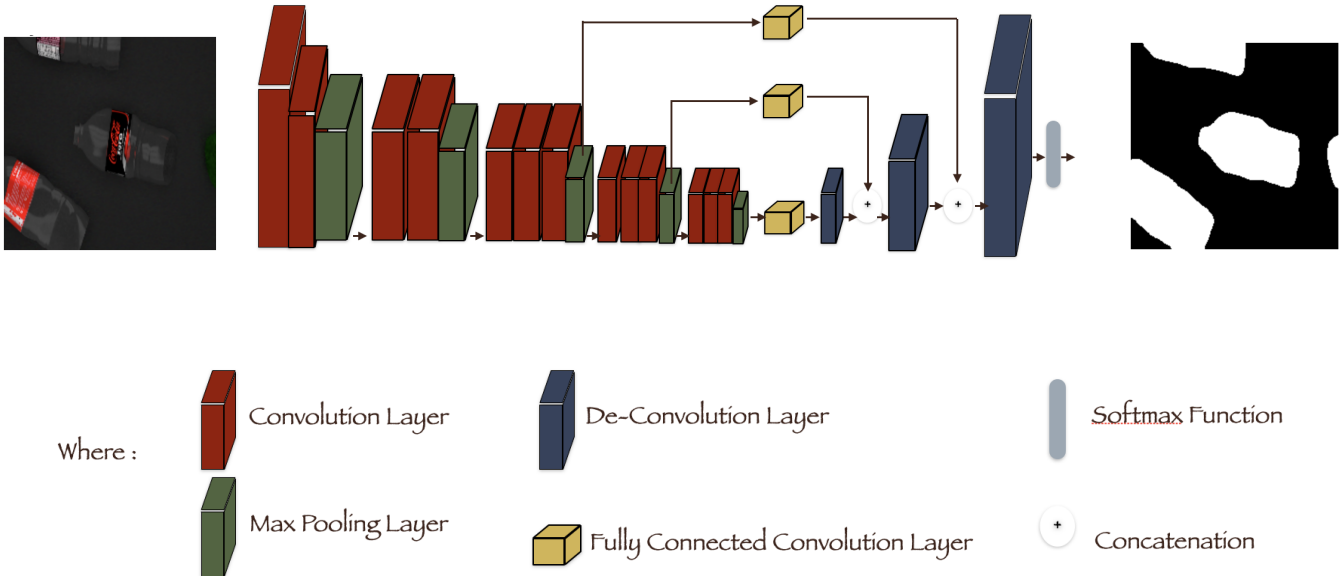


Fig 3: Architecture of the FCNN model used.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 224, 224, 3)]	0	[]
conv2d (Conv2D)	(None, 224, 224, 64)	1792	['input_1[0][0]']
conv2d_1 (Conv2D)	(None, 224, 224, 64)	36928	['conv2d[0][0]']
max_pooling2d (MaxPooling2D)	(None, 112, 112, 64)	0	['conv2d_1[0][0]']
conv2d_2 (Conv2D)	(None, 112, 112, 128)	73856	['max_pooling2d[0][0]']
conv2d_3 (Conv2D)	(None, 112, 112, 128)	147584	['conv2d_2[0][0]']
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 128)	0	['conv2d_3[0][0]']
conv2d_4 (Conv2D)	(None, 56, 56, 256)	295168	['max_pooling2d_1[0][0]']
conv2d_5 (Conv2D)	(None, 56, 56, 256)	590080	['conv2d_4[0][0]']
conv2d_6 (Conv2D)	(None, 56, 56, 256)	590080	['conv2d_5[0][0]']
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 256)	0	['conv2d_6[0][0]']
conv2d_7 (Conv2D)	(None, 28, 28, 512)	1180160	['max_pooling2d_2[0][0]']
conv2d_8 (Conv2D)	(None, 28, 28, 512)	2359808	['conv2d_7[0][0]']
conv2d_9 (Conv2D)	(None, 28, 28, 512)	2359808	['conv2d_8[0][0]']
max_pooling2d_3 (MaxPooling2D)	(None, 14, 14, 512)	0	['conv2d_9[0][0]']
conv2d_10 (Conv2D)	(None, 14, 14, 1024)	4719616	['max_pooling2d_3[0][0]']

```
conv2d_11 (Conv2D)              (None, 14, 14, 1024  9438208     ['conv2d_10[0][0]']
                                )

conv2d_12 (Conv2D)              (None, 14, 14, 1024  9438208     ['conv2d_11[0][0]']
                                )

max_pooling2d_4 (MaxPooling2D)  (None, 7, 7, 1024)  0           ['conv2d_12[0][0]']

conv2d_13 (Conv2D)              (None, 7, 7, 9)      9225        ['max_pooling2d_4[0][0]']

conv2d_transpose (Conv2DTransp  (None, 14, 14, 9)   1305        ['conv2d_13[0][0]']
ose)

conv2d_14 (Conv2D)              (None, 14, 14, 9)    4617        ['max_pooling2d_3[0][0]']

add (Add)                       (None, 14, 14, 9)    0           ['conv2d_transpose[0][0]',
                                                                 'conv2d_14[0][0]']

conv2d_transpose_1 (Conv2DTran  (None, 28, 28, 9)   1305        ['add[0][0]']
spose)

conv2d_15 (Conv2D)              (None, 28, 28, 9)    2313        ['max_pooling2d_2[0][0]']

add_1 (Add)                     (None, 28, 28, 9)    0           ['conv2d_transpose_1[0][0]',
                                                                 'conv2d_15[0][0]']

conv2d_transpose_2 (Conv2DTran  (None, 224, 224, 9)  20745       ['add_1[0][0]']
spose)

activation (Activation)         (None, 224, 224, 9)  0           ['conv2d_transpose_2[0][0]']

==================================================================================================
Total params: 31,270,806
Trainable params: 31,270,806
Non-trainable params: 0
__________________________________________________________________________________________________
```

Model Training:

The model is trained for 1000 epochs with 10% validation split from the training data. The batch size has been taken as 52. The model achieves a loss of 0.9051 and an accuracy of 0.8070. The model weights are stored and later used to plot graphs to highlight the performance of the model. Truncated training has been utilized to achieve optimal performance for this task. Plots for training loss vs iteration, training accuracy vs iteration, validation loss vs iteration and validation accuracy vs iteration have also been generated, which are as follows:

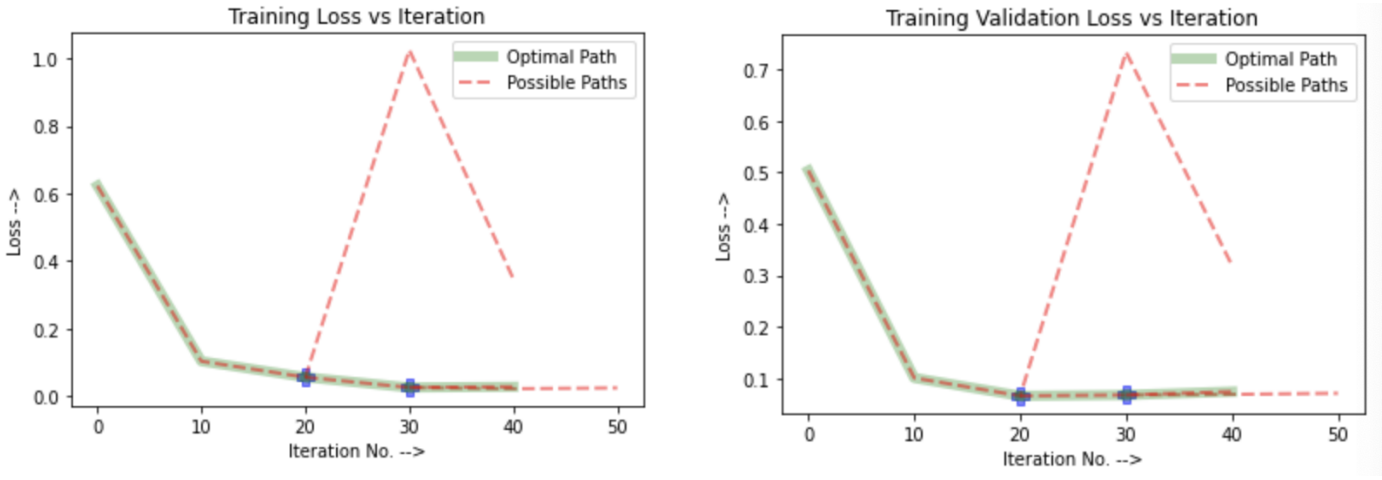


Fig 7: Plots for training loss vs iteration and validation loss vs iteration.

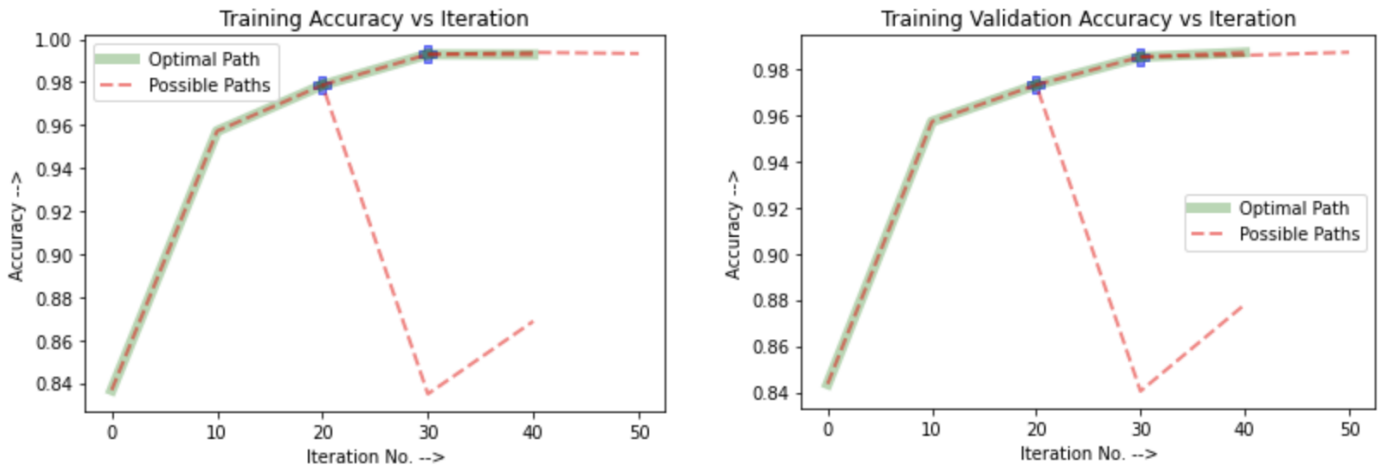


Fig 8: Plots for training accuracy vs iteration and validation accuracy vs iteration.

The model was first trained for 40 iterations but a spike in the loss and a fall in the accuracy was noticed. This was tackled by jumping back to the 20^{th} iteration by using the saved weights of the 20^{th} iteration using the callback, and from there the alpha_changer function is called which divides the learning rate by 10 after every 10^{th} iteration. This is performed twice for the training at points marked by the blue plus signs. The blue plus signs indicate checkpoints. This leads to the fall in the loss and an enhanced accuracy.

Image Reconstruction and Patch pasting:

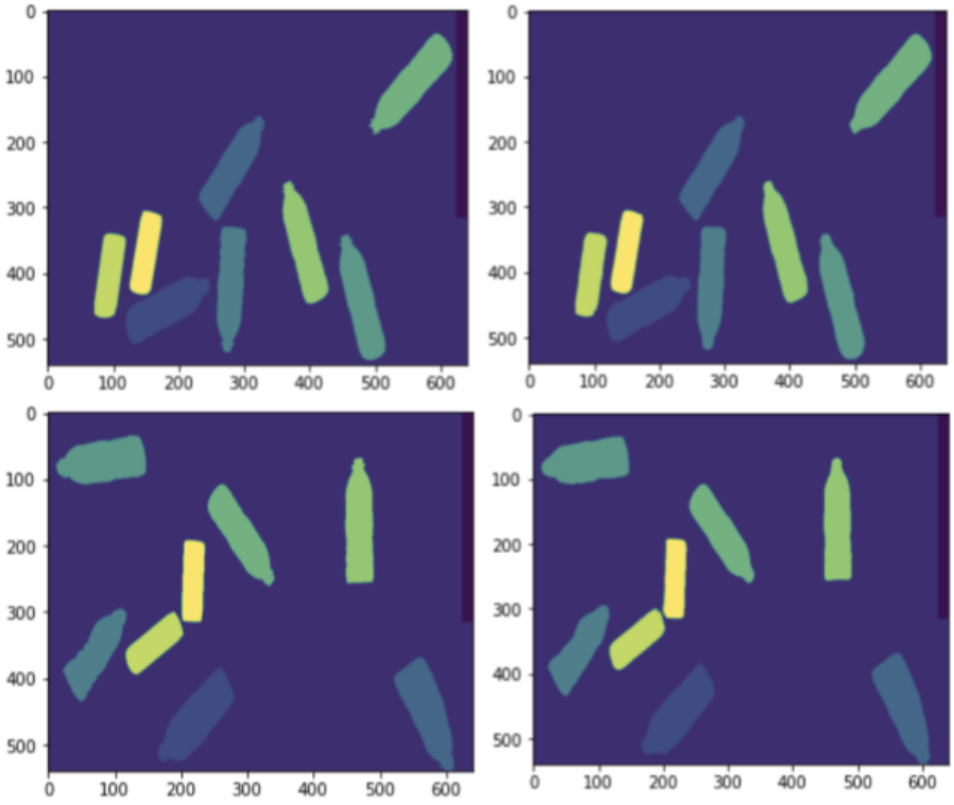
The output images are passed into the image_reconstruction function which uses the patch_paste function to paste the 26 patches of size 224x224 to reconstruct the patched image.

Model Testing and Evaluation:

The model is tested using the saved weights of the trained model and uses adam optimizer with learning rate 0.0001, categorical cross entropy loss function and uses accuracy as the performance metric. A loss of 0.09313610196113586 is obtained, while the accuracy obtained is 0.9878478050231934.

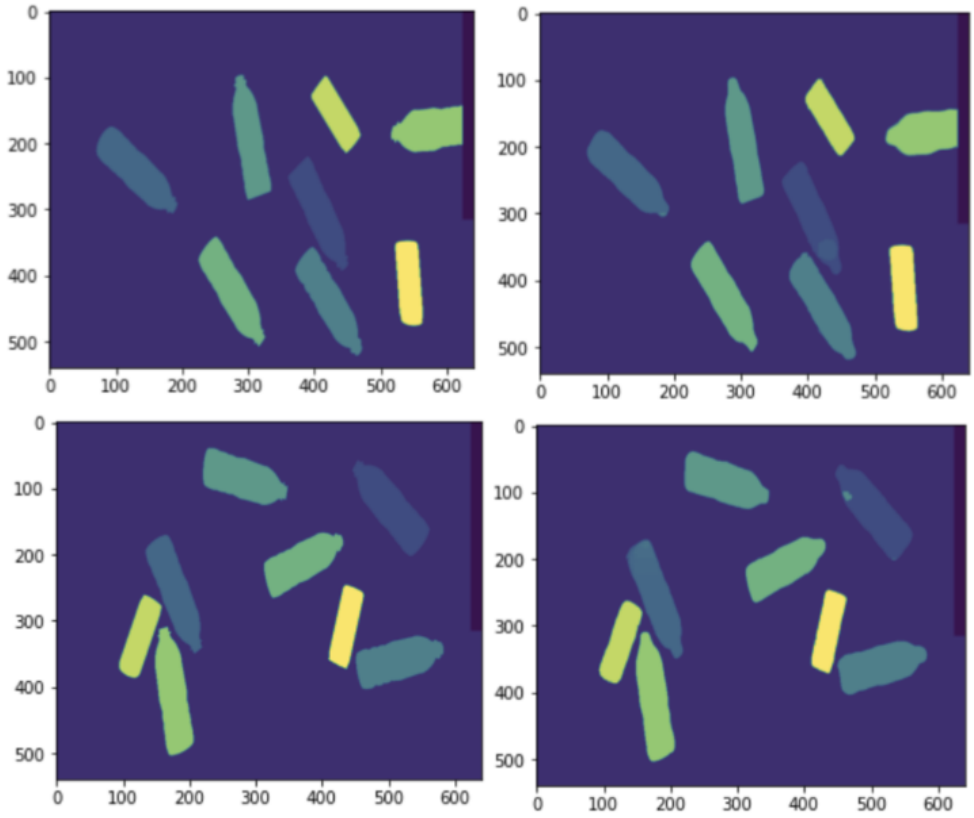
Results:

Below are several final (predicted + reconstructed) output images juxtaposed with their actual (ground truth) images, to showcase the model performance.



(a) (b)

Fig 4: (a) Actual VS (b) Predicted (Reconstructed) images.



(a) (b)

Fig 5: (a) Actual VS (b) Predicted (Reconstructed) images.

The predicted (patched) and actual (patched) images are given below.

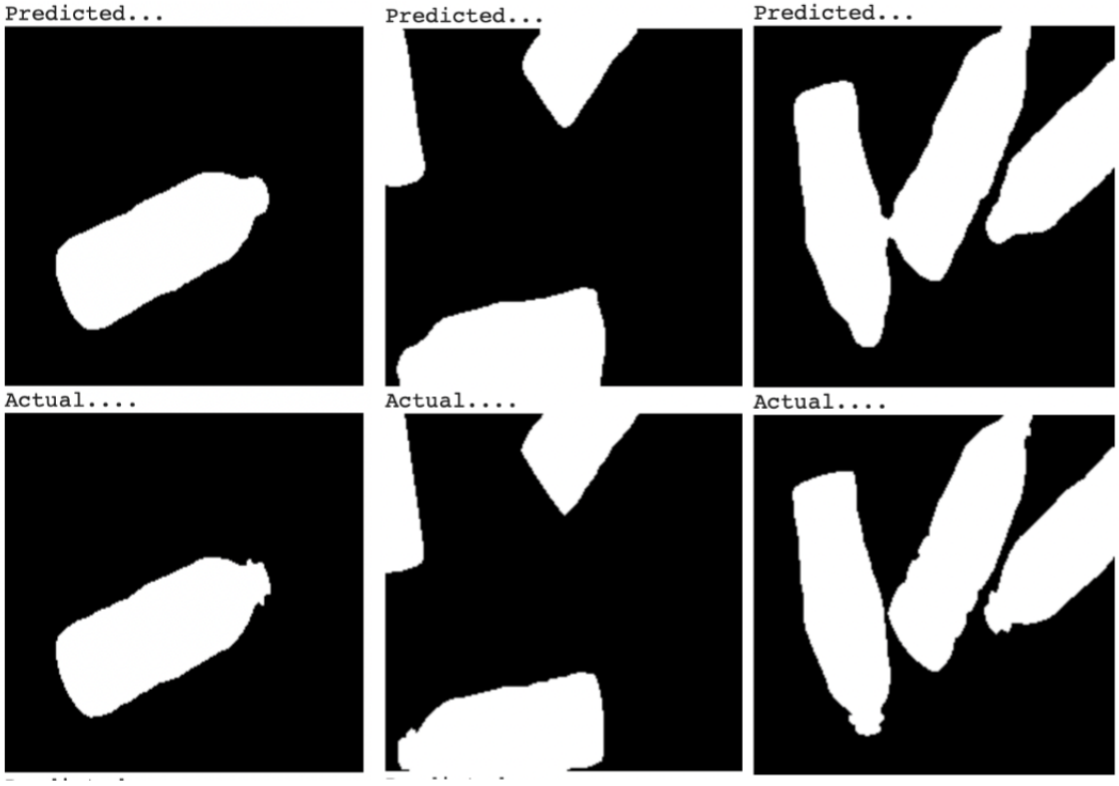


Fig 6: Predicted VS Actual (Patched) images.