Final Project
Image Inpainting using Autoencoder and Partial Convolution

Disha Nayar - 6199 1035 | Nayanika Ghosh – 4191 4976

Figure 1 : The left images show images with irregular holes. The right images the predicted results of our network on test data.

1. Abstract

Image inpainting is one of the popular techniques today to restore corrupted images. Given an image with holes, the image inpainting method learns to predict the missing portion of the image and reconstructs the complete image without holes. We have implemented a deep learning based solution for incorporating image inpainting. Our work is motivated by Image Inpainting for Irregular Holes Using Partial Convolutions [1]. Like [1], we have used partial convolution in place of convolution in our deep network. Our deep network follows a standard UNet architecture with encoders, decoders and skip connections between the first two. Unlike [1], we are generating the masked images and mask using OpenCV libraries. We have also used a L2 loss function for training our network. Our model is trained and tested on CIFAR10 images.

2. Introduction

Inpainting is a way of conserving damaged images to revive back it's missing part. The most common application of this method is in image editing like replacing an object with another or reviving old artwork. Most work in image inpainting is done by inducing rectangular squares in the image and training a network on the same. Inspired by [1], for this project we aimed to make irregular holes in the images and train our network to restore the original images. This was done to represent real life damages to images, as holes are never perfect rectangles. As in [1], we have implemented a model using partial convolution and autoencoders. Partial convolutions are implemented such that at every layer, the output only depends on the input values that are not holes. The method also incorporates automatic mask update at each layer which makes sure that eventually any mask will get filled up.

Since we are using CIFAR 10 dataset, we have changed the architecture so that training can be done on images of smaller sizes. Our model has 8 encoder layers and 8 decoder layers.

We tried different encoder-decoder architectures to get state-of-the-art performance and show two networks in our results. Our first network performed quite poorly, but modifying the number of layers, and the amount of upsampling and downsampling made us get state-

of-the-art results. We have further compared our model to other models trained on CIFAR 10 in section 5.4.

3. Related work

There has been numerous work done in the field of image inpainting. One of the methods is a Patch based method [2], where the model iteratively searches for patches in regions of images with no holes. The search for a relevant patch can also be done in other input images. Patch based methods are however not fast enough to be used for real time applications. There have also been approaches [5] that formulate image inpainting as an optimization problem. Even though the results obtained are good, these processes are generally slow.
Recently, many papers have also been written on image inpainting using GANs which give state-of-the are performance. However, GANs require large data for training and take more time to train. Our project is motivated by [1] and we manipulate the architecture to work with smaller images.

4. Technical description

Our complete system can be broken down into generating the correct data for training, designing and training the network, and testing our network. The input image from CIFAR10 dataset is first used to create a mask. This mask is designed in a way, such that when applied to the input image, it creates holes in the images. We call an image with holes as a masked image. Both the masked image and the mask are fed into the network for training. We follow an auto update rule for masks during the training process. The predicted image from our network is then compared to the input image to calculate the loss. We use L2 loss for our training. Figure 2 gives a diagram representing the complete flow.

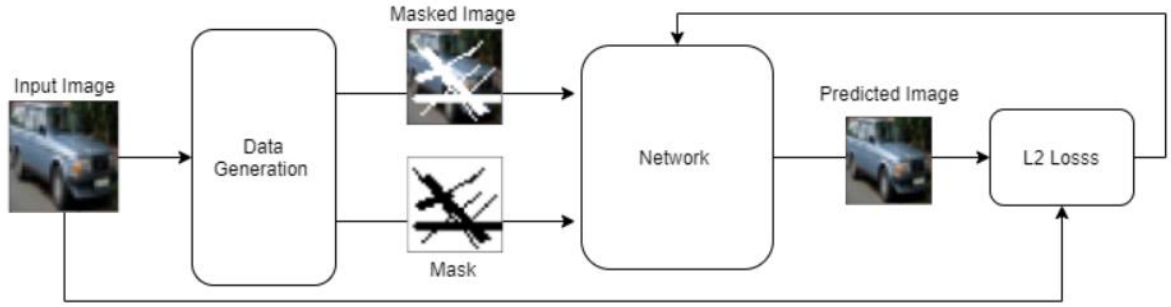

Figure 2: Workflow diagram of the system. Input image is fed to data generation to get the mask and masked image, which are then fed to the network to generate output. The predicted image from the trained model is then used to calculate loss along with the original input. Loss is used in backpropagation to update network weights.

4.1 Data Generation

Our input dataset is the CIFAR10 dataset which we directly download from pytorch torchvision library. The reason behind choosing this dataset is its size. The CIFAR10 dataset has 32x32 size images and are computationally less expensive for model training on a low resource machine. For images in the dataset, we are generating binary masks with random holes using OpenCV libraries. These masks are then applied to the input image to create the

corresponding masked image. The masked image, mask and the original input image are normalized to lie between 0 and 1 since we are using sigmoid in our network. Further, all three are then passed to the network for training and testing.

4.2 Network architecture

In the following section we first define partial convolution as it is an integral part of our architecture. We follow the explanation with our network design.

4.2.1 Partial Convolution Layer

For our implementation, a partial convolution operation followed by a mask update step forms the partial convolution layer. The mathematical formulation for partial convolution is given as in equation 1.

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

Where,
X = current features present in the window.
M = corresponding binary mask.
W = convolution filter weights.
$\odot$ = represents element wise multiplication.
sum(1)/sum(M) = scaling factor to account for the varying unmasked inputs.
b = bias.
Next, we update the mask. The mask is updated as in equation 2. If after the convolution, the output has at least one valid input, then the mask is updated to 1. That means that the input location will no longer be masked.

$$m' = \begin{cases} 1, & if\ sum(M) > 0 \\ 0, & otherwise \end{cases} \quad \text{eqn 2}$$

Hence, we can see that with multiple layers of a partial convolution, eventually any mask will get filled up.

4.2.2 Network Design

Our network design follows the Unet architecture with 8 encoder modules and 8 decoder modules. Each of the modules consist of a partial convolution layer, followed by batch normalization and a LeakyRelu. Batch normalization standardizes the input to a layer for each mini batch, thus removing the effects of internal covariate shift. This is also used to accelerate the training. We use a LeakyReLU to avoid vanishing gradient problems. Also, it overcomes the problem of dead neurons encountered while using ReLU. For encoder, one set of alternate modules have an increasing number of feature maps learnt and the next alternate set of modules are responsible for downsampling the image and the mask by setting the stride for partial convolution as 2. This architecture allows us to have more layers, unlike a regular per layer downsampling that limits the number of encoder layers to 4 for a 32x32 image. Similarly, in decoder, one alternate set of decoder modules is responsible for upsampling the image by scale factor of 2 and the next alternate set is responsible for reconstructing a lesser number of feature maps from higher number of feature maps. The modules that are responsible for upsampling takes a concatenated input from the previous decoder module

and the corresponding output of the encoder. Finally, at the end, the output received is the same size as the input image. Figure 3 gives the architecture of our network.

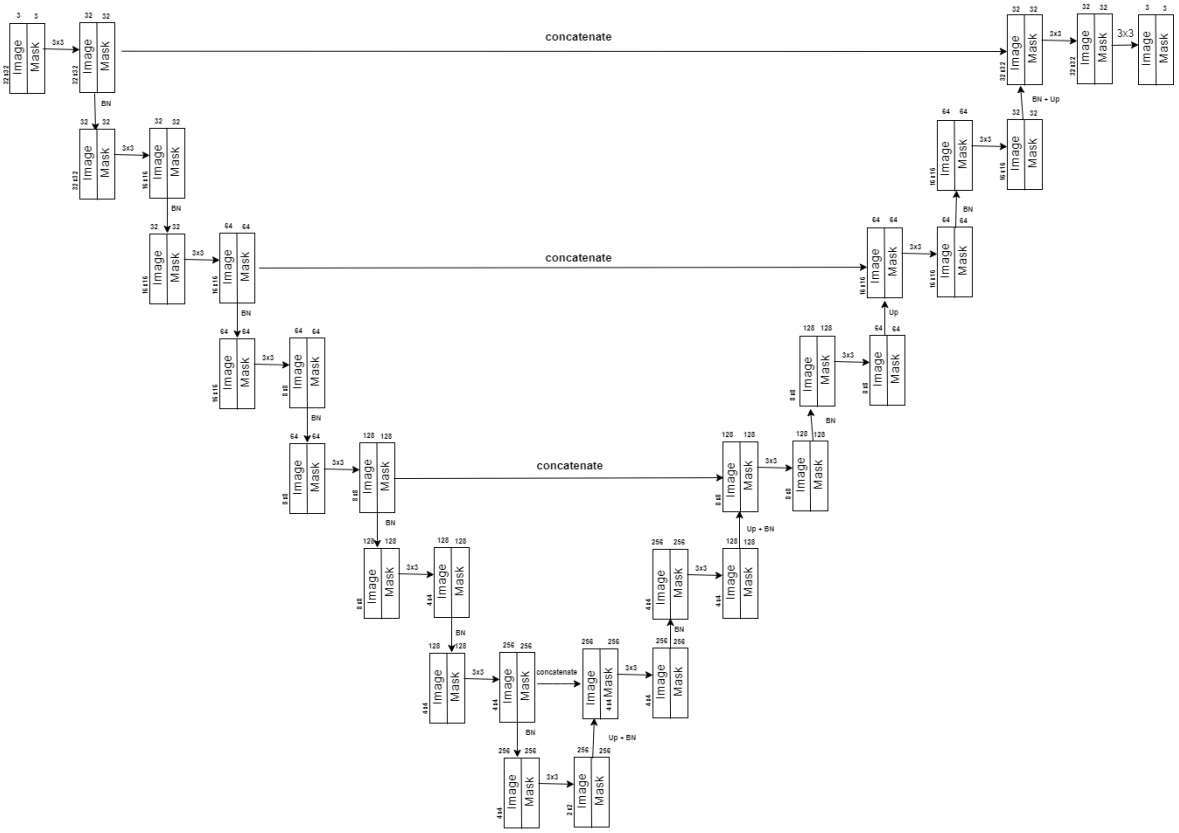

Figure 3: Network architecture of the system. The masked image and mask are fed into the autoencoder network consisting of 8 encoders and 8 decoders and skip connections labelled by *concatenate*

4.3 Training

In our implementation, the partial convolution layer handles both the masked image and the mask. Since we do not want any learning for masks, we set the weights for the mask to be 1 and disable the gradient learning for it. For the masked image, the initial weights are chosen using gaussian distribution. We have used 10,000 CIFAR10 training images. Our training data size is (32x32x3) which is fed to model in the batch size of 100 . For training the model, we are using an Adam optimizer with the learning rate of 0.01. Further the L2 norm is used as the loss function where the target image is the original input image. The network is trained for 100 epochs. We have also saved the trained model as *ImageInpainting.pt*

4.4 Testing

We use the pre-trained saved model *ImageInpainting.pt* to load the weights in the network. Testing is done on a set of 1000 test images from the CIFAR10 dataset. Prior to testing, we also set the model to evaluation mode by using model.eval(). We report the average loss over all the test images as a measure of performance.

5. Results and Analysis

5.1 Mask with irregular holes

For our implementation we needed irregular holes in our mask. We used open cv2 to create random holes of different width. Figure 4 shows a few images of how our mask looks.

Figure 4: The top image is the random mask generated. The black portion represents the holes. The bottom image shows the masked image, where the mask is applied to the input image.

5.2 Experiments and outcome

We show the results of the two architectures that we experimented with. The architecture layers of each of the two networks are given in table 1 and 2. We received state-of-the-art results for network 2. The training loss achieved for network is shown as a graph in Figure 5.

Encoder				**Decoder**			
Layers	**Downsample**	**In Channel**	**Out Channel**	**Modules**	**Upsample**	**In Channel**	**Out Channel**
Module_1	Yes	3	64	Module_1	Yes	128+64	3
Module_2	Yes	64	128	Module_2	Yes	256+128	128
Module_3	Yes	128	256	Module_3	Yes	512+256	256
Module_4	Yes	256	512	Module_4	Yes	512+512	512

Table 1: Network detail of network 1. The left side corresponds to the encoder with order from top to bottom. The right side of the table corresponds to decoder with order bottom to top

Encoder				Decoder			
Layers	**Downsample**	**In Channel**	**Out Channel**	**Modules**	**Upsample**	**In Channel**	**Out Channel**
Module_1	No	3	32	Module_1	No	32	3
Module_2	Yes	32	32	Module_2	Yes	32+32	32
Module_3	No	32	64	Module_3	No	64	32
Module_4	Yes	64	64	Module_4	Yes	64+64	64
Module_5	No	64	128	Module_5	No	128	64
Module_6	Yes	128	128	Module_6	Yes	128+128	128
Module_7	No	128	256	Module_7	No	256	128
Module_8	Yes	256	256	Module_8	Yes	256+256	256

Table 2: Network detail of network 2. The left side corresponds to the encoder with order from top to bottom. The right side of the table corresponds to decoder with order bottom to top.

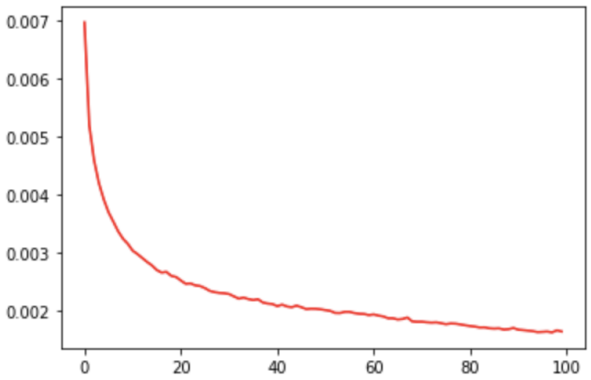

Figure 5: Training loss over 100 epochs for Network 2 trained on 10000 training image.

During testing, we received an average loss of 0.0023 for Network 2, whereas Network 1 gave us an average loss of 0.0094. The test results are shown for both the networks in figure 6. We noticed that with more layers, our model performs better.

5.3 Comparison

To analyse performance of our model, we have compared it to other state-of-the-art models. Since we are using a smaller size dataset (CIFAR 10), we wanted to compare our results with other models trained on the CIFAR dataset. Hence, we compare our results to the ones discussed in [3] and [4]. [3] discusses two state of the art methods. The first architecture is a fully convolutional network and the second is a convolutional network with fully connected layers. Results for both the methods are shown in figure 7. The results shown for the first architecture are as defined by Deep network architecture mentioned in [3]. The network consists of 5 convolution layers. The results for second network architecture are as defined by Fully Connected architecture mentioned in [3]. The network has 4 convolution layers

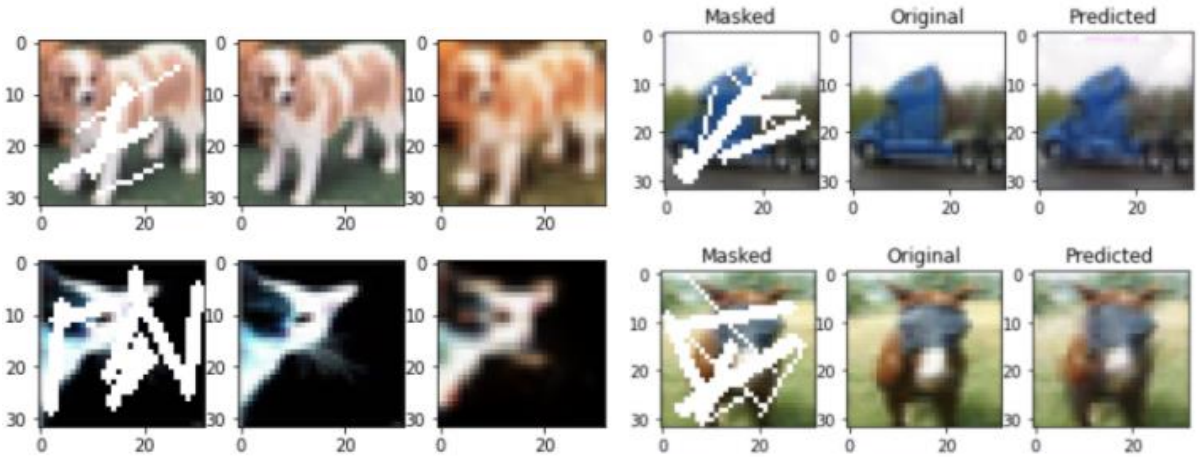

Figure 6: Test results for network 1(Left) and network 2(Right). The first image is the masked image. Second image is the ground truth and the third image is the model's predicted output.

followed by 2 fully connected layers. Next, we compared our model with the autoencoder architecture with vanilla CNN approach as in [4]. The output for the same is given in in figure 8.

Figure 7: The output images from [3]. The Left image is from a fully convolutional network and the right image is from a convolutional network with fully connected layers.

-

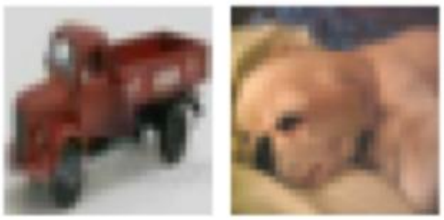

Figure 8: The output images from [4]

In the three architectures that we compared our model with, the holes in the images are rectangular or square regions. Our model is superior in this case as it accounts for irregular holes. We display our network output in figure 9.

Figure 9 : The output images from our model.

6. Conclusion and Limitations

In this project we have developed a deep network with partial convolution layers to restore an image with holes to its original state. The partial convolution helps to learn the missing part only from the valid part of the image and thus expected to be more robust than the convolution layer. Our network gave comparable results with respect to state-of-the-art architectures for CIFAR10 dataset. Since our network was designed keeping in mind the smaller image size of our dataset, our network does not give the best results for larger sized images.

7. Steps to run

Our submission consists of a zip folder with following entities :

1. 2 Jupyter notebooks - *ImageInpainting_train.ipynb* and *ImageInpainting_test.ipynb* for training and testing respectively.
2. Pre-trained model file (*ImageInpainting.pt*) that can be directly used for testing.
3. Results folder which consists of 4 sub folders. Mask, Masked Image, Predicted Image and Original image - Same image name in each folder corresponds to outcome of one image.
4. videotour folder – Consists of a video demo explaining and running our code.
5. Presentation Folder – Presentation.pdf represents the slides used for presentation and Presentation_video.mp4 is our presentation for this project.

<u>Steps to run each file</u>

1. Unzip ImageInpainting.zip
2. Training:
    1. Open ImageInpainting_train.ipynb
    2. This notebook consists of 4 sections - Data Loading and mask generation, Partial convolution implementation, Network Implementation and Model training.
    3. To run this file, click Run -> Run all cells
    4. The CIFAR10 training dataset will automatically get downloaded when the cell runs using torchvision library.
    5. To view the images predicted during training, scroll to the training section and see the images generated at each epoch.
    6. After the model has trained, you can view the graph depicting the training loss vs epochs.
3. Testing:
    7. For testing the Image Inpainting network, open ImageInpainting_test.ipynb. This file loads the ImageInpainting.pt pre trained model.
    8. To run the file, click Run -> Run all cells
    9. The CIFAR10 test dataset will automatically get downloaded when the cell runs using torchvision library.
    10. Scroll to the testing section to view the results

8. Reference

[1] Guilin Liu, Fitsum A Reda, Kevin J Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image inpainting for irregular holes using partial convolutions. In Proceedings of the European Conference on Computer Vision (ECCV)

[2] Barnes, C., Shechtman, E., Finkelstein, A., Goldman, D.B.: Patchmatch: A randomized correspondence algorithm for structural image editing. ACM Transactions on Graphics-TOG 28(3), 24 (2009)

[3] Mason Swofford. 2018. Image Completion on CIFAR-10. Computing Research Repository abs/1810.03213 (2018).

[4] http://cs231n.stanford.edu/reports/2017/pdfs/328.pdf

[5] C. Yang, X. Lu, Z. Lin, E. Shechtman, O. Wang, and H. Li. High-resolution image inpainting using multi-scale neural patch synthesis. arXiv preprint arXiv:1611.09969, 2016.