Image Processing - Image Inpainting Report

Wavid Bowman

April 2025

Forenote: Find references to what files hold different information in the methodology. There is also a README.md in the submitted repo with information to aid code readability.
Code can be found at `https://github.com/wavid-b/Img-proc-ec/`
or on HiPerGator at blue-woodard/wavid.bowman/img_proc_ec

1 Introduction

Image implantation is the process of reconstructing missing, obscured, or corrupted parts of images in a plausible way–usually in the "most" plausible way. Underlying concepts, both in traditional and modern methodologies. These include: ensuring local smoothness and continuity, self-similarity or the repetition of patterns in the image, mask-awareness, and maximizing both local and global consistency – i.e. the local texture/details must fit in and the global scene must fit; often this is done balancing pixel-wise loss vs a measure of global realism. Another important concept common in machine learning now is uncertainty, that there are multiple valid outputs in most scenarios. [4]

The basic operational principles are as follows:

- First, divide the image into a "known" and "unknown" region, representing the pixels that are likely to be uncorrupted and those that are corrupted.
- Next, estimate plausible values for the unknown region using the known region; this can be done using propagation where you extend local structures, or synthesis where you use global context and learned priors.
- Third, fill the image either all at once (most DL approaches) or iteratively from the edges of the unknown region (traditional/PDE approaches).
- Next, consistency enforcement is performed to ensure both local and global consistency; L2 loss is usual for local and adversarial or perceptual losses for global.
- Finally, repeat this process to create better results; most models are iterative, but there are some that are one-shot

[1] Provided information specific to traditional approaches, [7] provided information about DL approaches, and [4] provided an overview for the operational principles.

Limitations of image inpainting in modern methodologies include issues with large/complex missing regions. When the region is large, it becomes more difficult for models, especially DL models, to generate plausible content. It is also difficult to ensure global consistency, unlike local consistency. Where you can ensure local consistency using L2 loss, perceptual and adversarial-based loss sometimes struggle to ensure image look actually realistic. Finally, inpainting often fails to reconstruct fine textures like hair and grass convincingly. [11]

2 Methodology

2.1 Obfuscation

To perform image obfuscation, I created functions for generating masks for obfuscating regions of a 28x28 image with degrees of obfuscation ranging from 10% to 70% at 5% intervals. There are 3 functions, and for each level of obfuscation and each image, the function chosen was random. The first creates a spline-style obfuscation, where a random contiguous shape is generated with area based on the % needed to be obfuscated. The second creates a circle with a radius for best approximating the % obfuscation at a random location within the 28x28 array. The third creates a square or triangle at random as closely as possible to the area for % obfuscation required. Then, the masks were applied to the train and test images from the mnist dataset (retrieved via pytorch) to form the images with different levels of occlusion. All 70000 have a corresponding image at each level of occlusion. These masks were all created and applied using numpy.

A potential issue with this methodology is that the "corrupted" area may have not been a part of the number and as such not need to be inpainted, especially on lower-levels of corruption.

This section corresponds with img_splines.ipynb for the creation of masks and create_obfuscated.ipynb for applying them to mnist images.

2.2 Binary Classification Pipeline

For my Binary Classifier, I am using a simple linear kernel SVM. The basic pipeline is as follows:

1. Normalize and denoise the data before feature extraction. To denoise the data, I performed a median filter, then a mean filter, then clipping to ensure that it is still in the range 0, 1. This should remove the worst of the noise from the image if there is any.

2. Extract features to be given to the SVM. The features extracted include raw pixel information, the global histogram, and edge features (mean edge

strength and total edge strength). It concatenates these features to single flattened array per pixel.

3. Next, features are scaled using a standard scaler and passed to the SVM for training.

The SVM had a C of 1 and linear kernel for simplicity of implementation and time complexity concerns.

The code for this section can be found in binary_classifier.ipynb.

2.3 Noise

To better evaluate the machine learning pipeline, sources of noise were added to to both the "non-corrupted" and "corrupted" images before training and evaluation. The sources of noise I added were Gaussian ($\mu = 0, \sigma = 15$), salt and pepper (both 1% probability), and Poisson noise. There is one pipeline trained and evaluated on the noisy images and one on the non-noisy images; their architectures and pipelines are identical.

Application of noise was done in add_noise.ipynb

2.4 Corruption Localization Pipeline

In order to implement a Corruption Localization pipeline, I utilized YOLO, a deep learning architecture [10]. First, I obtained true bounding boxes from the masks I created during the occlusion task. These served as the labels to be tested against in the learning process of YOLO, where a CNN trains on data, comparing against the true labels with a loss function, and eventually validating to ensure correct performance. This architecture automatically extracts model parameters and feature representations. I trained with 3 epochs, upscaling the mnist-based images to 64x64.

To generate the bounding boxes from masks, go to gen_bounding_box.ipynb. For the test/validation/train split, go to split_files.ipynb. To run the detection algorithm, activate the conda yoloenv and run the detect script in the yolov5 directory. This is modified from the default yolo to only return the most confident bounding box, as there is at most 1 region of corruption.

2.5 Inpainting Algorithm

In order to generate the best inpainting algorithm for this project, I compared the results of 4 different algorithms: an extension of my localization pipeline using using Telea's algorithm, one using LaMa's fully convolutional neural network to perform inpainting, one using zits' transformer-based approach and one Criminisi-style patches. [9] [8] [2] [3] . Due to the issues with calculating well localized corruption region, I have run these with masks from the true boxes so that issues with level 2 do not strongly impact performance at level 3. This skews the results some, but with better object detection and localization, the following pipeline can be used the same way.

I used openCV's implementation of Telea's algorithm, and created a simple patching algorithm for my implementation of Criminisi patching. The code for this part can be found in patch-approach.ipynb.

For using Lama, activate the conda environment partialconv and run the lama_inpainting.py script in the main folder of the repo. For this, I use the lama architecture using the lama-cleaner API [8] [5]. To change to zits transformer model or a latent diffusion model, change the arguments of config's model field and model's name field accordingly.

I also used the lama-cleaner API to use the ZITS transformer-based approach [2].

3 Results

3.1 Binary Image Classification Pipeline

The overall accuracy of the Binary Classification Pipeline was at 0.9156 without noise and 0.9128 with noise. This is relatively good performance for a vanilla machine learning approach.

3.2 Corruption Localization Pipeline

I reached an average IOU of 62.66% in testing. Other performance benchmarks include a precision of 0.989, recall of 0.989, mAP50 of 0.991, and mAP50-95 of 0.52.

3.3 Inpainting algorithm

All of the analysis and results for this section are calculated in section3.ipynb.

3.3.1 Telea's algorithm

The pipeline using Telea's algorithm's average SSIM was 54.55%, decreasing from ¿90% to close to 30%, and had an MAE of 0.08

3.3.2 Patching algorithm

The patch-based approach had marginally better performance than Telea, achieving an average SSIM of 55.88%. It had an average MAE of 0.0761.

3.3.3 LaMa Algorithm

The LaMa FCNN approach performed better than both traditional approaches; it had an average SSIM of 59.91% and MAE of 0.0736.

3.3.4 ZITS Algorithm

The final and best-performing approach was a ZITS transformer-based approach. It reached 60.45% SSIM and an MAE of 0.0766.

4 Discussion

4.1 Binary Image Classification Pipeline

4.1.1 Impact of Noise on Classification Pipeline

There was little impact of noise on the overall performance of the Classification pipeline. The overall accuracy was at 0.9156 without noise and 0.9128 with noise; a minute change.

4.1.2 Impact of Obfuscation on Classification Pipeline

As you can see in figure 1, at lower levels of obfuscation, the accuracy of the SVM is considerably worse. Eventually, it reaches perfect accuracy at 50% obfuscation both with an without noise. This is likely related to the issue I mentioned in the methodology, where the corruption may not affect the object at lower obfuscation levels, only the background.

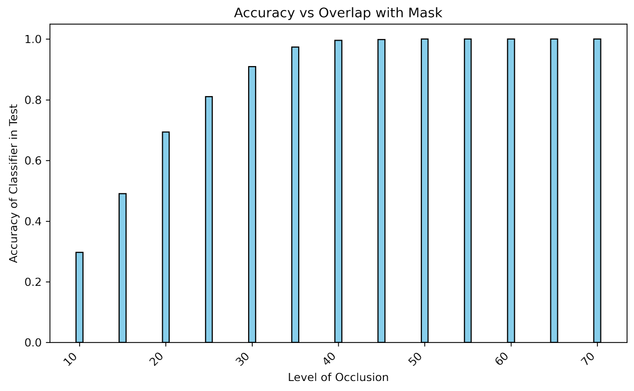

Figure 1: Impact of Level of Obfuscation on Accuracy

The type of obfuscation also greatly impacts the model's ability to recognize corruption, as seen in figure 2. Performance is relatively level for circles and polygons, but splines–the randomly formed occlusion region–are most difficult to detect

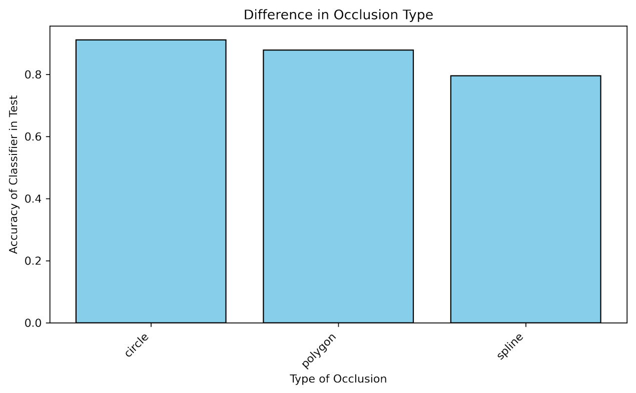

Figure 2: Impact of Level of Obfuscation on Accuracy

4.2 Corruption Localization Pipeline

From the high accuracy and precision at low IOU thresholds but low average IOU, I can conclude that the corruption localization pipeline I implemented is very good at recognizing corruption, but not good at localizing it. With more training and a different error function more focused on localization, this would likely be possible.

Figure 3 shows how the performance increases with more occlusion until it slowly becomes worse.

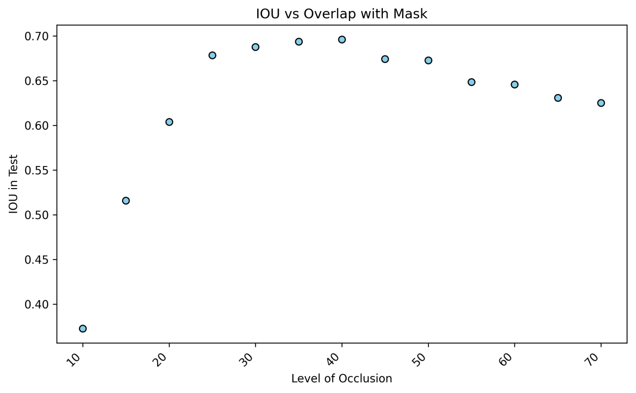

Figure 3: Impact of Level of Obfuscation on IOU

I also computed how much of the true corrupted region was contained in the calculated box; this was 78.83% globally, and the values separated by occlusion level can be found in figure 4. This shows how the performance is best at the middle-levels of occlusion, but at low and high levels it performs poorly.

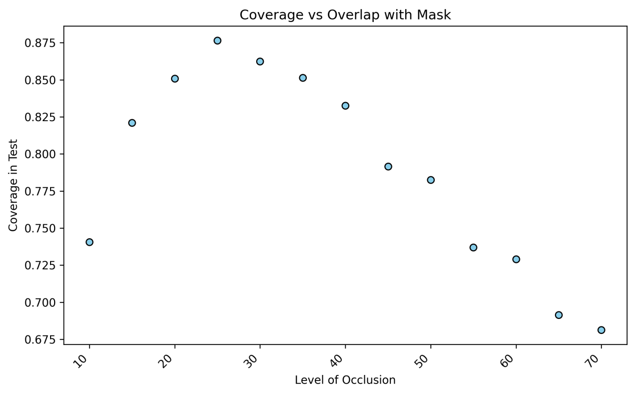

Figure 4: Impact of Level of Obfuscation on Coverage

These calculations were done by the YOLO algorithm during runtime, except for IOU which is found in IOU calculation.ipynb

4.2.1 Failed approaches and insights: Corruption Localization

I attempted to continue training after the first round of tests. However, it started to fail to recognize when there was a lack of corruption and its IOU and precision fell to ¡75%, its IOU also falling. In future directions, it may be better to try a different architecture for localization. Overall, I would prefer better performance in corruption localization, but section 3 is more important.

I also made a mistake during my first training of the model, not including any non-corrupted images. This caused issues with the model overestimating the amount of corruption, even in image with some corruption.

4.3 Inpainting Algorithm

4.3.1 Telea's Algorithm

Telea's algorithm for inpainting resulted in a good MAE but had lackluster results for SSIM–especially as the level of occlusion increased. See figure 5 below for the chart.

Blue is 1-MAE and red is SSIM. IOU is not a valid metric here because I used the true masks for this data (see Methodology).

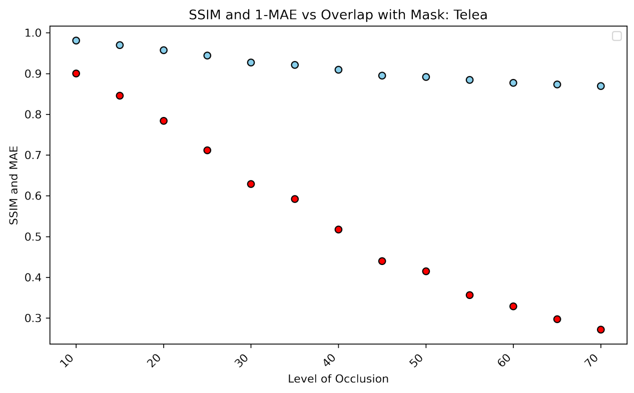

Figure 5: Telea Inpainting results

4.3.2 Patching Algorithm

This algorithm performed better than Telea's, but it's another example of the classical algorithms failing to perform well. Again, blue is 1-MAE and red is SSIM. See figure 6.

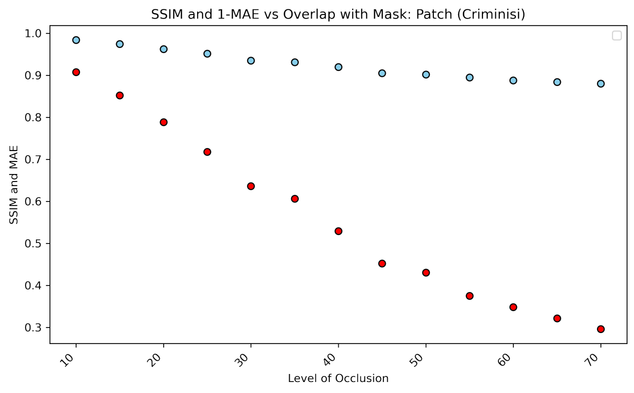

Figure 6: Criminisi Inpainting results

4.3.3 LaMa Algorithm

The LaMa approach performed better than the traditional methods, but failed to reach benchmarks due to a lack of fine-tuning. With more See figure 7 for the SSIM (red) and 1-MAE (blue) as level of occlusion changes

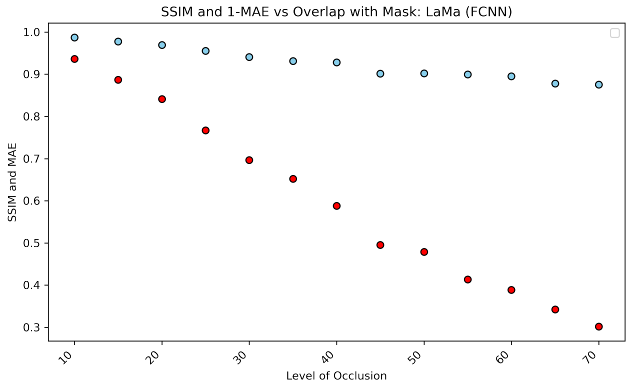

Figure 7: LaMa FCNN Inpainting results

4.3.4 ZITS algorithm

The final and best-performing approach was a ZITS transformer-based approach. Again, performance could be better with more fine-tuning, but it met some minimum benchmarks. Figure 8 shows the same types of changes in SSIM and 1-MAE as the other approaches.

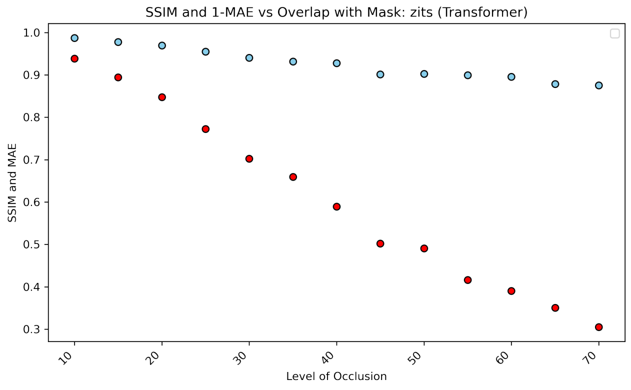

Figure 8: ZITS Inpainting results

4.3.5 Failed approaches and insights: Inpainting

I also attempted to use partial convolutional neural networks for inpainting, but I could not produce good results using the NVIDIA PartialConv2d model [6]. I believe this is a direction that would take more time to flesh out. I also tried an out-of-the-box latent diffusion model through the lama-clean API, and it performed very poorly.

All of these approaches had the same issues–their performance became drastically worse as overlap with the mask increased. This is expected, of course, as occlusion prevents more of the image from providing context.

The traditional methods performed far below the deep-learning approaches to inpainting. The time I spent implementing and running those experiments could have been better spent on the deep-learning methodologies in retrospect. I also had issues fine-tuning the weights for the ZITS model and the LaMa model; when I attempted to fine-tune them the weights would not run with the script I had built using the Lama-clean python API. With more time, compute, and effort, maybe these could have been better.

5 Results

My best overall results come from the ZITS transformer-based impainting algorithm. It had a final average SSIMs of 60.45% overall, with 93.9% average SSIM in the 10% occlusion case. It also had less than 10% MAE at every level of occlusion, with an overall MAE of 0.0766.

References

[1] Marcelo Bertalmío, Guillermo Sapiro, Vicent Caselles, and C. Ballester. Image inpainting, 01 2000.

[2] Chenjie Cao, Qiaole Dong, and Yanwei Fu. Zits++: Image inpainting by improving the incremental transformer on structural priors, 2023.

[3] A. Criminisi, P. Perez, and K. Toyama. Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on Image Processing*, 13(9):1200–1212, 2004.

[4] Omar Elharrouss, Noor Almaadeed, Somaya Al-Maadeed, and Younes Akbari. Image inpainting: A review. *Neural Processing Letters*, 51(2):2007–2028, December 2019.

[5] Victor Gu. lama-cleaner: A backend inpainting tool powered by various sota models. `https://github.com/Sanster/lama-cleaner`, 2021. Accessed: 2025-04-29.

[6] Guilin Liu, Kevin J. Shih, Ting-Chun Wang, Fitsum A. Reda, Karan Sapra, Zhiding Yu, Andrew Tao, and Bryan Catanzaro. Partial convolution based padding. In *arXiv preprint arXiv:1811.11718*, 2018.

[7] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting, 2016.

[8] Roman Suvorov, Elizaveta Logacheva, Anton Mashikhin, Anastasia Remizova, Arsenii Ashukha, Aleksei Silvestrov, Naejin Kong, Harshith Goka,

Kiwoong Park, and Victor Lempitsky. Resolution-robust large mask inpainting with fourier convolutions, 2021.

[9] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of Graphics Tools*, 9, 01 2004.

[10] Ultralytics. yolov5 github, April 2025.

[11] Hanyu Xiang, Qin Zou, Muhammad Nawaz, Xianfeng Huang, Fan Zhang, and Hongkai Yu. Deep learning for image inpainting: A survey. *Pattern Recognition*, 134:109046, 09 2022.