Image Inpainting with Basic Convolutional Networks

Robin Meneust, Ethan Pinto

December 2024

1 Introduction

In the context of our "AI-Based Image Processing" course, we worked on this project, in which we reproduced and tested a specific image inpainting approach, defined by the paper *"Context Encoders: Feature Learning by Inpainting"* (Pathak et al., 2016)[1].

Image inpainting consists of filling hole(s) in an image. There exist different methods to do so (e.g. they compared their results with Photoshop). In this paper, they used a context encoder trained in an adversarial way. Basically there is a generator, this is our context encoder (here an encoder and a decoder) that given an image of size 128x128 with a dropout region (a "hole", with values set to 0) tries to predict what should be inside the hole. We focused on the simplest case here for the dropout region: a square in the center of size 64x64 (i.e. half of the image). This is a large section of the image, so the task is complex. There is also a discriminator that tries to predict if an image is false (generated) or true (the real region that was dropped). It's also important to note here that there are no pooling layers, because it's "detrimental for reconstruction-based training" as they mentioned in their paper. So we only have convolution layers with batch normalization and activation functions.

The authors use two loss functions: reconstruction loss and adversarial loss that they combine using a weighted sum (of parameters λ_{rec} and λadv): this is the joint loss. The reconstruction loss is basically a MSE (Mean Squared Error) loss. A model using only this loss will output blurry images because it minimizes the mean pixel-wise error. It will be clearly visible in the section 4 and in our GitHub[1] in the results images. That's what the adversarial loss will fix. Indeed this loss will "encourage the entire output of the context encoder to look realistic, not just the missing regions". So it will be sharper than with the reconstruction loss, as we can see in their paper in the figure 1. It's basically BCE (Binary Cross Entropy) loss. Note here that the encoder only considers the masked region (dropout) and not the whole image. This isn't really clear in the loss section of the paper, but it is on their model architecture image, here in figure 2, since the discriminator input is 64x64 while the full image is 128x128.

In this report, we will first explain how we implemented the model as it's defined in the paper in section 2.

[1] https://github.com/Hanabi-TheFox/Image-Inpainting-with-Basic-Convolutional-Networks

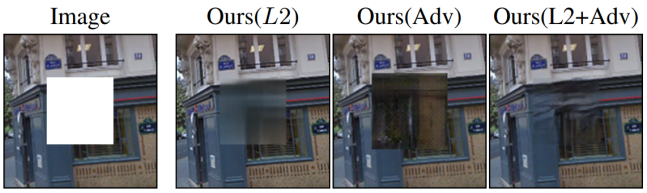

Figure 1: Comparison of Inpainting Results Using Rec, Adv or Joint Loss (Results From the Paper)

Then we will describe our experiments setup (datasets, hyper-parameters...)3. Finally, we will present and interpret our results for the tests that we conducted in the section 4. Those tests include dataset variations and different λ values. Our code will be available at a later date on GitHub (some time after the report deadline).

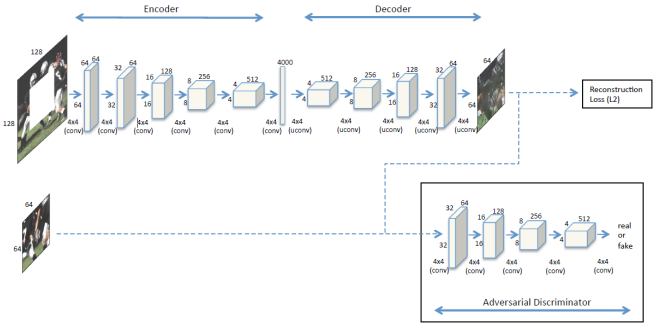

Figure 2: Model Architecture of Pathak et al. (2016).

2 Methodology

In this section we will focus on the coding environment and implementation details.

2.1 Development Environment

This project was developed on Python using PyTorch Lightning. This library simplifies the creation and use of PyTorch models by providing a high level abstraction and separating the model logic from the training loop. We used both PyCharm and Visual Studio Code for the IDE. We first set up a Conda environment to manage dependencies and simplify the installation process. The environment setup was automated using a YAML configuration file (environment.yaml) at the root of this project.

2.2 Python Package

Additionally, we built our project as a Python package (named image_inpainting). We had to write a pyproject.toml file to configure it. Defining a package allows the project to be imported and used as any other Python package. This is especially useful for imports in the code or if we want to use the project from another folder.

2.3 Datamodule and Dataset

In this section we define two types of classes: datamodule and dataset. A datamodule is a Lightning class used to manage dataloaders (for training, test and validation). And a dataloader object uses a dataset. A dataset is a PyTorch class used to fetch data from an associated dataset. In our case we defined it ourselves to choose custom datasets and a custom format for inputs and "labels" (the dropped out regions).

Some datasets PyTorch objects can download the datasets by themselves, but for big datasets we need to do it ourselves and specify the path when we initialize it. Once we have the images we apply transformations on them when they are fetched. They are first resized to 128x128, which is the input dimension for the model. Then they are converted to tensors and normalized to $[-1, 1]$, that is because this is the range of the output of the generator model used here (it uses tanh as we will see further). We compose all the transformations and we pass the transformation object to the associated dataset object.

Then, the dataset object will list its image file paths. When *getitem* is called, it will apply on the fetched image the transformations and extract the dropout region defined as a 64x64 square in the center of the image (simplest case), that is half of the image. When an item is fetched from the dataset, we return an input image, that is the image with zeroes in the dropout region, and the true dropout region, that is the part that was replaced by zeroes. This is the input and "label" used by the model.

2.4 TensorBoard

We used TensorBoard to visualize the training process in real-time. This is especially useful to stop the training early if the metrics are bad and compare the runs. Saving and loading the results is really easy and we even added an option to save some generated images per epoch (from the validation set). This is what the code in the callback on_validation_end does. Then we can see it in TensorBoard or generate a GIF of the evolution of the model outputs using the notebook written for it. We can also save our experiments' hyper-parameters and see it inside.

2.5 Metrics

2.5.1 PSNR

The paper uses PSNR so we also used it to make comparison easier. This stands for Peak Signal-to-Noise Ratio. The higher the value, the better it is, as it indicates that the reconstruction quality is good. It's understandable since MSE appears in a denominator in this metric[2]. We used torchmetrics here.

2.5.2 Adversarial Loss

Then we need the adversarial loss. It's defined in the paper as

$$\mathcal{L}_{adv} = \max_D \mathbb{E}_{x \in \mathcal{X}} \big[\log(D(x)) + \log\big(1 - D(F((1-\hat{M}) \odot x))\big)\big],$$

So in terms of loss we are minimizing the opposite. We can simply write this using BCE (Binary Cross Entropy) loss whose definition is : $-(y \log(p) + (1-y)\log(1-p))$, where y is the label and p the prediction. Here the adversarial loss is defined as $(BCE_{true} + BCE_{false})/2$. Because if y = 1 (true) then we have $BCE_{true} = log(p)$ where $p = D(x)$ (prediction of the discriminator that the true dropout image region is true). For BCE_{false} it's the same with $y = 0$ and $p = D(F(1 - \hat{M}) \bigodot x)$, that is because using the notation of the paper we are here getting the prediction of the discriminator for the generator output (i.e. fake input). We sum and divide by 2 because the paper mentions that it's an average ($\mathbb{E}$).

2.5.3 Reconstruction Loss

Here we first defined this loss as a L2 distance as mentioned, so it wasn't normalized and the loss was way too big, which was not good for training. So we checked their Lua code, and they used in fact MSE. This aligns with what is commonly used in this case, and with what they add in the paper when they say it "minimizes the mean pixel-wise error". Hence we computed MSE here (we used L2 distance and then we averaged it), but we need to consider a variation here. The paper mentions here that they increase the loss value by a factor 10 in the overlapping region: that is a 7px wide area on the borders of the dropout region. It is used to make the prediction more consistent with the context. To do so, we computed the l2 distance in the overlapping and non overlapping region (using a binary mask). Then we summed those two using the 10 factor for the overlapping one.

2.5.4 Joint Loss

This is simply a combination of the two losses above in a weighted sum. In the paper, the adversarial loss has a small weight $\lambda_{adv} = 0.001$ compared to the reconstruction loss with $\lambda_{rec} = 0.999$. It's important to note that

[2] PSNR

the paper is not clear here. They seem to use here the adversarial loss mentioned above, but instead they use only the BCE_{false} part, that is the usual approach in GAN, and also what they have done in their Lua code[3]. This is because the generator doesn't care whether the discriminator is good at identifying true images. It wasn't to know how bad the generator prediction is, if it's easily identified as false or not.

2.6 Model

Our package have different sub modules. Loss defines the losses for our model (reconstruction loss...). Utils defines helper functions especially used to print results. Dataset and datamodule modules have the dataset and datamodule classes for the different datasets defined in 3. Finally, model defines the different models and the Lightning module class. Note that we used the same model as defined in the paper[1], so for more details please read it and look at the figure 2.

We first created four PyTorch modules for: encoder, decoder, generator and discriminator. This improve the modularity of our project and its readability.

The generator is composed of an encoder and decoder. The encoder maps an input image to a latent vector of size 4000, while the decoder reconstructs the original image from this latent representation. Note that the channel-wise fully-connected layer is not used for the square dropout, as we can see in the figure 2, so we didn't implement it. Note that the output layer of the decoder layer has a tanh activation function (so we normalize the output to $[-1, 1]$), as explained in the cited paper[2].

The discriminator is a convolutional neural network that evaluates whether an image is real or generated (fake). On the output layer there is a sigmoid activation to output a binary classification score.

We then define our context encoder in a Lightning module to manage training, validation and testing loops on the generator and discriminator. This defines the overall structure explained in the paper. It's also where we use TensorBoard to save the metrics during each epoch. It also handles the optimizers. Here we have one for the generator and one for discriminator. Those two use Adam with parameters defined in section 3

2.7 Training Loop

When we used Lightning in our AIIP classes, we used automatic optimization (we didn't have to write the backpropagation steps). But here, especially since we use two optimizers, we need to disable it and do it manually[4].

During training we need to update both the generator and discriminator. So we need to toggle or untoggle the optimizers when needed, as mentioned in the Lightning documentation. First we update the discriminator using the output of the model using the loss in the section 2. Then we update the generator.

The rest of the process is the usual one: we compute other metrics if needed (here PSNR) and we log those in TensorBoard.

3 Experiments

The experiments were conducted on a RTX 3060 with 12GB of VRAM, 32 GB of RAM and the CPU was a Ryzen 9 7900X. We will describe here the datasets, the parameters for the training procedure, and other experimentation details. We used notebooks to run those tests, since it allowed us to both show an usage example of our package and keep the results saved directly in the notebook (images...). In our experiments we vary: the dataset and the loss coefficients λ.

3.1 Datasets

For our experiments, we used two datasets:

- **Tiny ImageNet**[5]: This dataset consists of 200 object classes, each containing 500 training images, 50 validation images, and 50 test images. Thus, there is respectively: 100,000 training images, 10,000 validation images, and 10,000 test images. The images are of size 64×64 pixels and represent a wide variety of objects, including animals, vehicles...
- **ImageNet-1k (ILSVRC 2012)**[6]: This dataset includes 1,000 object classes with around 1,000,000 training images, 50,000 validation images, and 100,000 test images. The images vary in size and depict a diverse range of objects and scenes. We first cropped and resized this dataset to 128x128 with a Python script. We also created an other version in 64x64.

We began our experiments with Tiny ImageNet because of its smaller size. It allowed us to make more tests in a shorter period of time and was adapted to our hardware. Once the preliminary experiments were successful, we moved on to the larger dataset, to further evaluate our approach and provide more examples so that our model can get better at generalizing.

3.2 Hyper-parameters

The default hyper-parameters are taken from the paper[1], the cited paper[2] and their Lua code. Note that the number of epochs is not in the following list as we used different one in our runs (we often stop early while looking at the TensorBoard logs in real-time).

- Learning rates: 0.002 for adversarial and 0.02 for the generator (context encoder)

[3] Lua code of the paper
[4] Disable Auto-optimization
[5] Tiny ImageNet
[6] ImageNet-1k

- Batch size: 64 or 512 (the results didn't change much)
- $\lambda_{rec} = 0.999$ and $\lambda_{adv} = 0.001$
- Adam betas coefficients: 0.5 and 0.9

4 Results

In this last section, we will finally present our experiments results. We will first compare the test PSNR values in section 4.1 and then we will look at the generated images and discuss the results4.2.

4.1 Test PSNR

Table 1: Variation of The $\frac{\lambda_{rec}}{\lambda_{adv}}$ Ratio

Data (approach)	Size	Ratio				
		×1	×100	×200	×500	×1000
Tiny ImageNet (Ours)	64 × 64	11.56	14.34			**14.76**
ImageNet-1k 64 (Ours)	64 × 64					**17.20**
ImageNet-1k 128 (Ours)	128 × 128		13.17	13.40	**15.39**	14.70
Paris StreetView (Original paper)	128 × 128					**18.58**

The results in 1 first indicate that our implementation has worse performance compared to the initial paper. However, we need to note that the dataset is different (they didn't provide PSNR results for ImageNet). Additionally, using their recommended parameters (especially λ_{rec} and λ_{adv}) is not always the best choice, as our tests on ImageNet-1k have shown. It will be especially visible in 4.2. We can also note that using a 64x64 image and rescaling it to 128x128 gives better performance for ImageNet. This can be because PSNR considers the difference in quality between the real and reconstructed images, so if the real one is of poorer quality the difference is lower. It might also be because it doesn't have to be as precise, there are fewer sharp details to reproduce. It's very important to note that the results here are for different epochs. We just picked the model with the best validation PSNR value across all epochs and we tested it to get those results.

4.2 Generated Images

Note that you can see in our GitHub animated images showing the evolution of the model outputs across epochs. There are also more results in the notebooks, since we can't be exhaustive in this report.

Figure 3 shows that with default parameters the generated image is quite good considering half of the image is missing. However, it's too blurry, so we tried increasing the weight of the adversarial loss. The results are in figure 4. Here there is almost no blur and the results are very similar to the ones in the paper, even though it's not perfect.

The two others are on Tiny ImageNet. We tried setting the same weight for adversarial and reconstruction

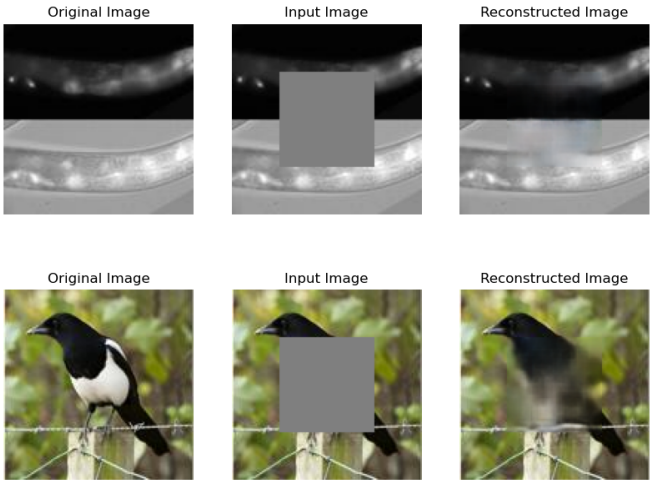

Figure 3: Experiment on ImageNet 128x128 with Standard Parameters (×1000 Ratio)

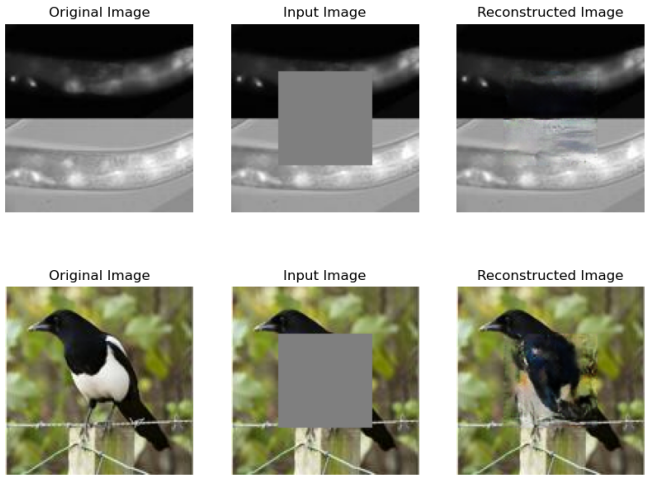

Figure 4: Experiment on ImageNet 128x128 with ×200 Ratio

loss for once in figure 5. The results were to be expected, and are aligned with the original paper results (on adversarial loss only). The context doesn't seem to be taken into account, the results are not blurry but are totally off compared to what we want. Note that compared to ImageNet-1k, we obtained quite good results with the parameters of the paper in figure 6.

No matter the experiment, we always observed some errors on especially difficult images. That is for example those with many colors, small items, lots of details...

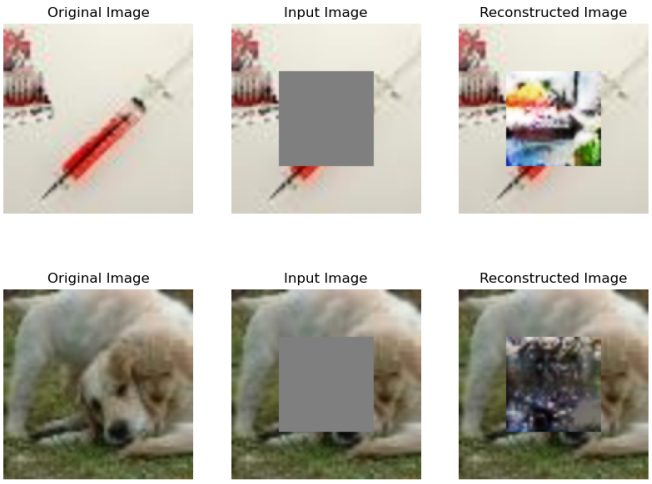

Figure 5: Experiment on Tiny ImageNet with $\times 1$ Ratio

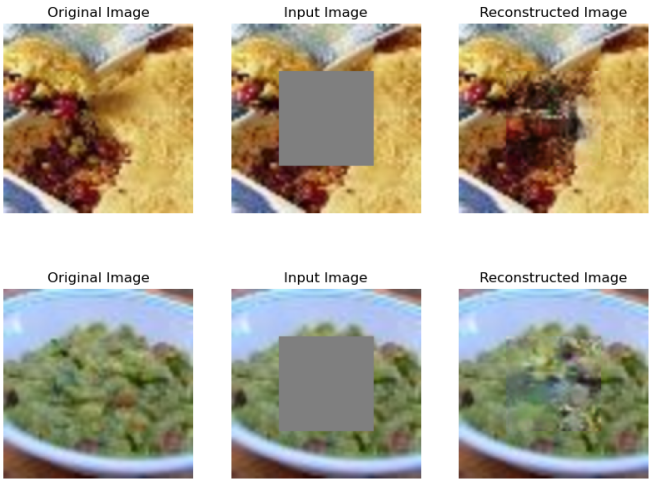

Figure 6: Experiment on Tiny ImageNet with Standard Parameters ($\times 1000$ Ratio)

5 Conclusion

Our results are not as good as the initial paper, that can be due to the difference in dataset. So we might want to consider other simpler datasets. We should also consider changing other parameters such as the learning rate. We can then add noise, as they suggest themselves. We only considered the simplest case as asked for this project. But using pre-trained models as the paper did with AlexNet (when the dropout region is not a square) might also improve our results. In this project we provided a PyTorch Lightning implementation of a context encoder in a simple Python package to facilitate the understanding of the paper and run experiments on its model architecture. We also added tools to make the visualization of the results easier. We typically added an option to save images per epoch and create an animated image out of it.

References

[1] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," 2016. [Online]. Available: https://arxiv.org/abs/1604.07379

[2] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016. [Online]. Available: https://arxiv.org/abs/1511.06434