Test-Time Training for Image Inpainting

by

Genseric Ghiro

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Abstract

Image inpainting is the task of filling missing regions in images with plausible and coherent content. The usual process involves training a CNN on a large collection of examples that it can learn from, to later apply this knowledge on new, unseen images with areas to complete. One important consideration is that each training dataset has its own domain, and domains in the image space are extremely diverse. For optimal results, the user image at inference time should belong to the same domain as the training examples, which greatly limits the range of inputs a trained network can be used for. Moreover, collecting new data is hard, and training on different datasets requires a great amount of computation power and time. In this thesis, we propose a test-time learning approach for inpainting. More specifically, we train a CNN like described above, but in addition, we use the user image with holes to build a new dataset, on which we continue training the network for a small number of iterations during test-time training. To the best of our knowledge, test-time training has never been done for inpainting. With this technique, our hope is that the model will learn to fill holes that are specific to the user image better. It also facilitates domain adaptation, which means that a wider range of input domains at inference time can produce acceptable results when using the same pretrained model. We obtain results demonstrating that even for a state-of-the-art model, our method can achieve significant improvements, in particular for perceptual scores. Moreover, if used in a software, it has the advantage of being optional, meaning that the user can choose to keep the original result if there is no significant improvement, which makes our framework beneficial in all situations.

Acknowledgements

I would like to thank my supervisor, Professor Olga Veksler, for giving me the opportunity to study under her supervision, and for all her continuous support and help that have made this thesis possible.

Table of Contents

List of Figures

Chapter 1

Introduction

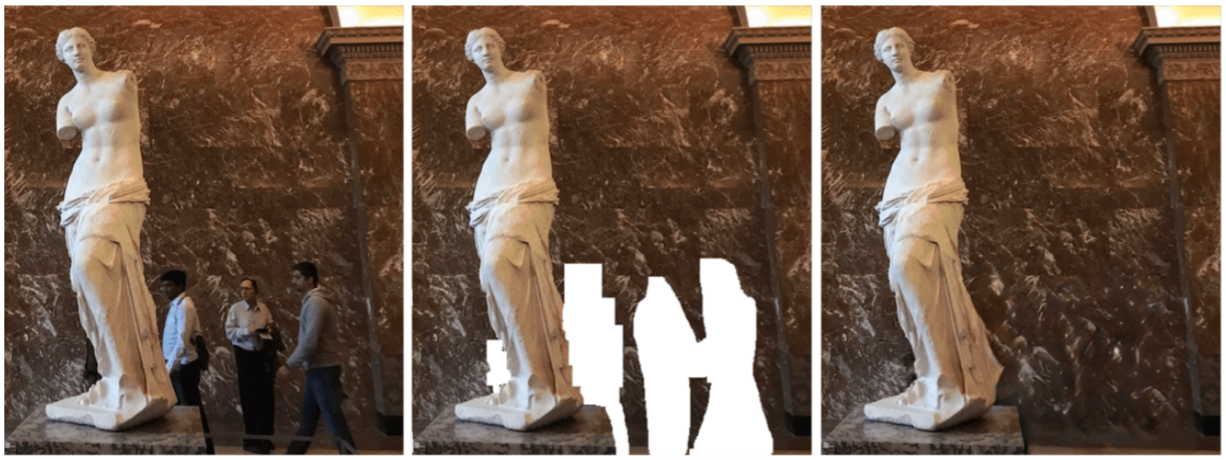


Figure 1.1: Inpainting example, taken from [35], figure 1.

Inpainting belongs to the domain of computer vision and is the task of filling holes in images. More precisely, given an image, it aims to replace pixels indicated by the user with new values that are consistent with the rest of the image. The concept of inpainting appeared in the 18th century, when museums started being concerned with restoring damaged paintings and ancient texts. However, it is only in the early 2000's that it gained more traction with modern techniques based on diffusion [1] or patch filling [5]. Another major breakthrough occurred in the 2010's when convolutional neural networks (CNNs) [17] achieved promising results. Since then, the amount of published work in this field has multiplied, and today, CNNs largely dominate the podium of state-of-the-art results.

1.1 Applications

Inpainting possesses a very wide range of applications, from object removal to super-resolution. Although each problem might seem unrelated, they all boil down to completing hidden pixels in an image, which can be achieved by strategically adapting the mask to each situation. Indeed, users can designate regions they would like the network to complete with a binary mask, where 1's, for example, represent the corrupted area. Therefore, text and object removal can be done by drawing a mask that covers the text or object that should be replaced. Similarly, for image restoration, damaged portions to restore should be highlighted by the mask. To a certain extent, inpainting can also be applied to denoising, when the noise regions can be explicitly designated. In this case, the mask needs to be accurate at the pixel level, and is used to indicate values that have the strongest noise.

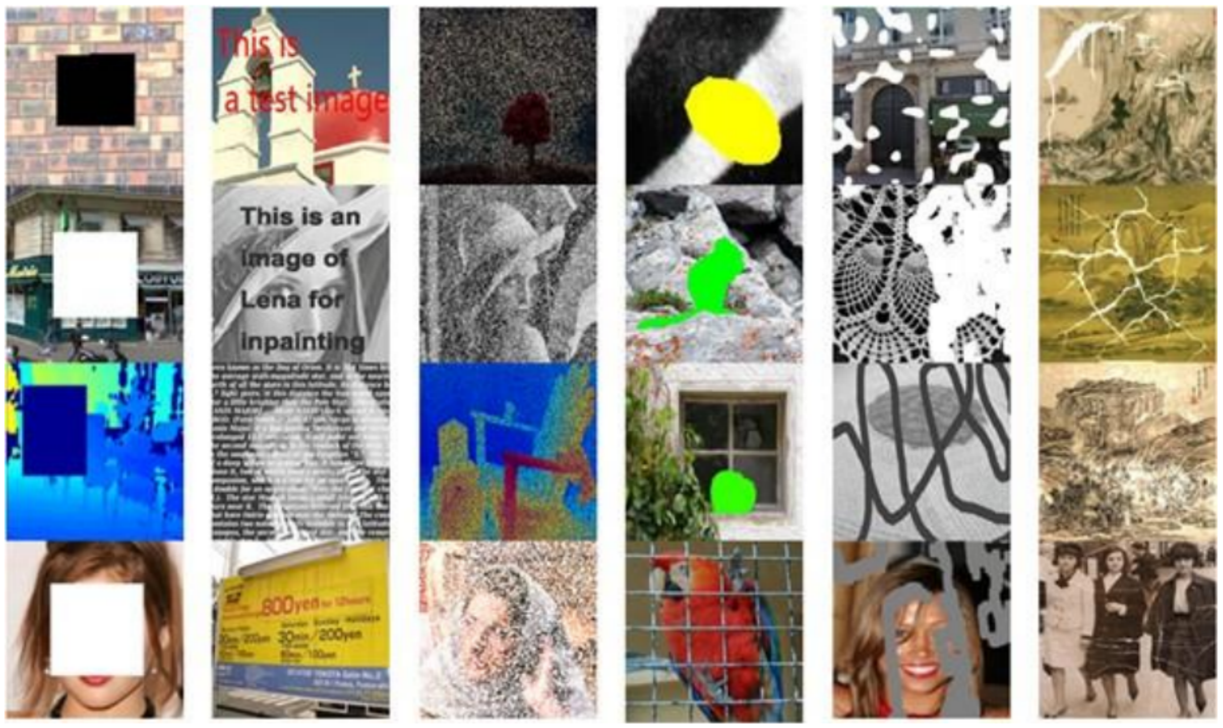
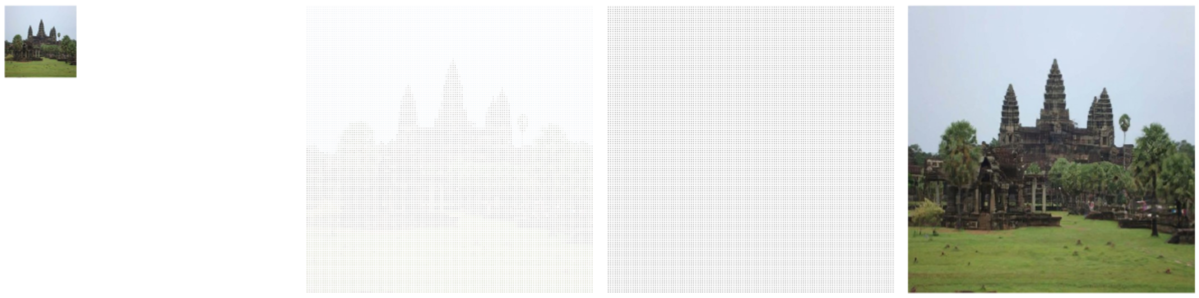


Figure 1.2: Example of image inpainting applications, taken from [7], figure 1. Represented applications are, from left to right, filling square regions, text removal, denoising, object removal, filling free-form regions, and image restoration.

Another, less conventional, but interesting, application of inpainting is super-resolution. This problem deals with upscaling an image while preserving, or even enhancing, its clarity and sharpness. There exist many algorithms and techniques for this task, and inpainting can be applied to the problem by ingeniously drawing the mask. The steps for this process are as follows. Considering a small image, all of its pixels can be distributed on a wider surface such that they preserve their spatial order, while being equally distant from each other. If the upscaling factor is large, this operation will produce a barely distinguishable image, since pixels will be sparse. Figure 1.3(b) shows an example of this. Finally, the mask is formed from the combination of all the regions between scattered pixel values (1.3(c)). Formulated this way, super-resolution becomes an inpainting problem, and can produce much clearer results than if an interpolation method had been used.



a) Low-resolution image b) Input c) Mask d) High-resolution result

Figure 1.3: Super-resolution with inpainting, taken from [19], figure 12.

While inpainting can be generalized to a great variety of tasks by carefully drawing the mask, it does, however, often require great precision that can be hard to automate with a computer. It makes sense for users to meticulously create masks that are tailored to a specific use case. However, training CNNs requires a large amount of examples, which is why most of them are trained on problems that do not necessitate manual labelling, like filling square regions and free-from shapes, and later applied to more specific tasks like text removal or denoising.

1.2 Training and evaluation

The training process for an inpainting model is straightforward since it usually does not require any manual labelling. Once the dataset is selected, random portions of each image are hidden by generating binary masks. Then, corrupted images and their respective mask are stacked together and fed to the network. The mask is also part of the input to explicitly indicate regions that are to be completed by the model. Finally, the original image, before being tempered with, is used as the ground truth when computing the loss function.

Besides training, evaluation is a crucial component of deep learning because it quantifies results, not only allowing models to be compared, but also guiding the network's learning. It is, nonetheless, one of the biggest challenges in inpainting, and what makes it such a difficult problem. Unlike most computer vision tasks, there is often more than one possible solution. For example, when trying to remove an undesirable object from the background, there can be many plausible and visually pleasing ways of filling what goes behind it. Moreover, in this situation, the ground truth does not provide any clue, since it contains the object to remove. On one hand, this makes the task of the network easier because the solution is flexible, but on the other hand, evaluation becomes harder because there can be several good outputs.

Normally, results are evaluated by comparing the output to the ground truth, and measuring the distance between the two, which encourages the model to produce a result that is as similar as possible to the original image. In inpainting, this is done with losses like L1 and L2, that consist in taking the sum of absolute differences or squared differences at the pixel level. However, because of the non-deterministic nature of inpainting, metrics that only evaluate the fidelity of the result with the ground truth are not sufficient, and even, sometimes, misleading. For this reason, perceptual evaluation is often preferred. This method aims to evaluate the plausibility of an image from a human perspective. In other words, it tries to quantify how likely a human is to believe that the image is real. Therefore, even if the result is dissimilar to the original input, it will not affect the score, as long as it is credible.

Even though perceptual metrics and losses are not based directly on a pixel-to-pixel comparison with the ground truth, they still utilize the original images by extracting high-level features, and comparing them to those of the inpainted result. Indeed, only a fraction of the input is masked, and high-level features are expected to be mostly consistent even if the output result is slighlty different. The original images can also be used to approximate the underlying data distribution that represents the training examples. Using this knowledge, a discriminator [8], presented in section 2.4, can then try to distinguish fake from real samples, which helps orient the network's training.

Finally, inpainting is a non-deterministic problem, which means that it can have more than one valid solution. Consequently, traditional forms of evaluation that insist on faithfully reproducing the original image may not be appropriate. Perceptual evaluation, on the other hand, puts the emphasis on human perception and plausibility, which is well-suited for multi-solution problems. Nonetheless, each type of evaluation can be preferred in different contexts. Indeed, for image restoration and denoising, the output should be as close as possible to the ground truth, while for object removal, it is the complete opposite.

1.3 Pluralistic inpainting

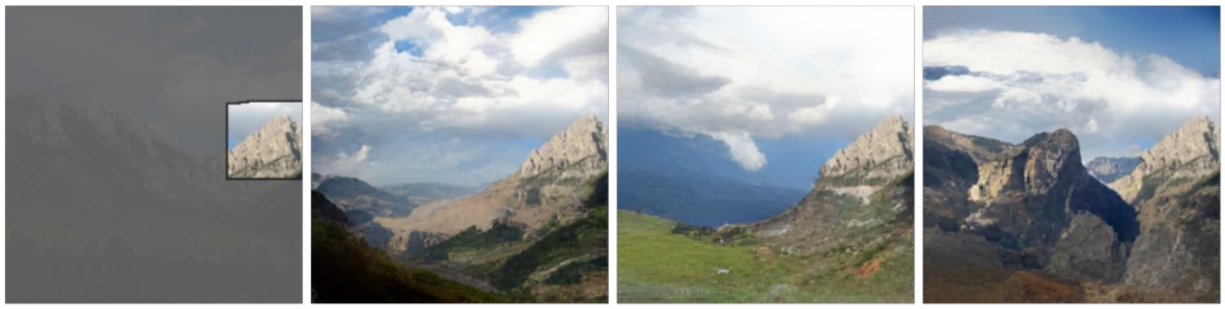


Figure 1.4: Sample outputs from a pluralistic inpainting model, taken from [30], figure 1. On the left is the input, where the dark region corresponds to the masked area. The three other images on the right are completed outputs from the network.

More often than not, inpainting networks are designed to produce a single result. However, as described above, inpainting is a pluralistic problem, since there can be many acceptable answers. This is especially true when a large portion of the image is masked. In this case, the valid context is not as restrictive, which grants more freedom to complete the hidden area. Figure 1.4 shows an example, where only a tiny square of valid context is preserved. Based on what it contains, the final output should probably feature a mountain on a cloudy blue sky background; however, there exist many ways of representing this as shown by the three results on the right.

Pluralistic inpainting, also called probabilistic inpainting, describes the task of outputting various completed results, when given a single image and mask as input. To produce many results, conditional prior features of the image, set by the valid context, are coupled with a random sample from a given distribution, such as a multivariate Gaussian distribution, which provides the randomness needed to generate diverse images. The decision of building a pluralistic model rather than a deterministic one (single output) is based on the context and expected use case. For example, when training with large masks, or when giving the user multiple choices is important, pluralistic models should be preferred.

1.4 Contributions

In this thesis, we propose a method, applied to inpainting, to leverage the valid context of the user input at test time, that we refer to as test-time training. Given a CNN trained for this task on a large dataset, and a user image with holes, we propose to finetune the given inpaining CNN for a small number of iterations on a dataset constructed from the single image given by the user. Our motivation is that the user image itself likely contains areas not covered by holes that are similar to the regions masked by the user. Therefore, finetuning on a small dataset constructed from the user image will adapt the trained CNN to the specific textures and structures present in the user image, making it more suitable for filling holes in that particular image. We explain how to adapt standard loss functions for image inpainting to our test-time training setting, in order to make the network focus on learning only on the valid regions of the input image, i.e. regions that are not masked by the user.

We also extensively test our framework and show that it is able to improve performance of a state-of-the-art network. To test our framework, we run multiple experiments using different assumptions and sets of parameters. Doing so, we are able to obtain a framework that learns quickly, and that can be adapted to many different networks and types of images.

1.5 Thesis organization

The first section of Chapter 2 gives a general overview of convolutional neural networks, how they work, and their core components. The second part of Chapter 2 outlines important work related to inpainting and the main breakthroughs of the last years. In Chapter 3, we describe our finetuning framework, and the motivation behind it. Chapter 4 presents how we tested this framework through extensive experimentation. For each experiment, we explain what it aimed to test, discuss the results, and, show sample outputs at the end. Finally, Chapter 5 is a summary of our work in this thesis, and it concludes with a few possible ideas for future work.

Chapter 2

Related Work

In this chapter, we will describe the related work.

Existing algorithms for image inpainting can be divided into two main categories: sequential, and convolution-neural-network-based methods. While sequential methods, the set of inpainting algorithms that do not make use of convolutional neural networks (CNNs), have set the foundations for image inpainting in the early 2000's, they were quickly outperformed by CNNs a few years later.

Sequential-based methods can be further subdivided into diffusion filling [1] and patch filling [5]. These approaches are built on two major ideas that form the core of image inpainting. First, the valid pixels around the masked region can give valuable clues as to how to complete it, and second, similar information to the one that is missing can often be found somewhere else in the valid region of the same image. Nonetheless, for more complex structures, and for missing regions that are different from the rest of the image, which is often the case, these two assumptions will not hold, and consequently, diffusion and patch filling will fail.

CNNs made their appearance in image inpainting as a more powerful alternative to diffusion and patch filling because of their impressive ability to learn from examples. In other words, even though similar information cannot be found in the valid region of the same image, they are able to learn from other similar images, and extend that knowledge to the current image. They are also good at exploiting the context given by the valid regions surrounding the masked area.

However, the intrinsic difficulty of the problem, and the complexity of the underlying image distribution made it hard for vanilla CNNs to achieve adequate results. An important key to solving this problem revealed itself when Goodfellow et al. invented the Generative Adversarial Networks (GANs)[8]. GANs are specifically designed to faithfully approximate complex data distributions, thus becoming an essential tool in image inpainting with CNNs.

Still, it was quickly realized that CNNs and GANs had a great weakness that was crucial to inpainting: long-range dependencies. Where patch filling methods succeeded so well, CNNs and GANs were not particularly good at borrowing distant information from the same image. In other words, even if the same repetitive pattern in the missing region could be found in the opposite corner of the image (in the valid region), it was not trivial for these algorithms to borrow that information. This led to another breakthrough in image inpainting, contextual attention [34]. The main purpose of this mechanism is to give CNNs the ability to explicitly borrow distant information in the same image.

In the next sections, we will first summarize how basic CNNs work, and then take a deep dive into the core components of image inpaiting such as GANs, contextual attention, and gated convolutions.

2.1 Convolutional neural networks

Convolutional neural networks belong to the family of artificial neural networks. They represent one of the many models in deep learning, and are mainly used with images. They were first introduced by LeCun et al. in 1989 in an attempt to classify hand-written digits [18], but it was only later in 2012 that their usage became more widely adopted with AlexNet, a deep learning model used for image classification [17]. This delay was largely due to the fact that CNNs, just like artificial neural networks in general, perform well only when given a lot of data. However, computers in the 90's and early 2000's were not fast and powerful enough to cope with this enormous amount of data that would make CNNs the powerful tool it is today. Their breakthrough, in 2012, can be attributed to their success for image classification, the task of categorizing an image into different labelled categories. However, they quickly emerged as the best approach in almost all cases for all types of image-related problems, from classification to inpainting.

The prefix "convolutional" comes from the similarity with discrete convolutions in mathematics. CNNs mainly comprise convolutional layers, which are functions that take 2D or 3D grids of numerical values as input. The output will vary according to the parameters of the layer, as well as the purpose of the model in which it is used. All the details will be made explicit in the following sections.

2.1.1 Convolutional Layers

At the core of convolutional layers lies the convolution operation. Simply put, this operation corresponds to the sum of the elementwise product between two matrices, a filter and an equal dimension window in the given input. More formally, let's define G, a 2-dimensional input, F, a $m \times n$ filter, and O, the resulting output. Then, a convolution can be expressed as:

$$O_{x,y} = \sum_{i=-k_h}^{k_h} \sum_{j=-k_v}^{k_v} G_{x+i,y+j} \times F_{i,j}, \tag{2.1}$$

where x and y are the center coordinates of a given window in G, $k_h = \frac{m-1}{2}$, and $k_v = \frac{n-1}{2}$. It is assumed that the center coordinates of F are (0,0) and that negative filter indices are allowed. The equation above assumes that m and n are odd. If m and n were even, k_h and k_v would respectively become $\frac{m}{2}$ and $\frac{n}{2}$, and the lower bounds of the summations would instead be $-k_h + 1$ and $-k_v + 1$.

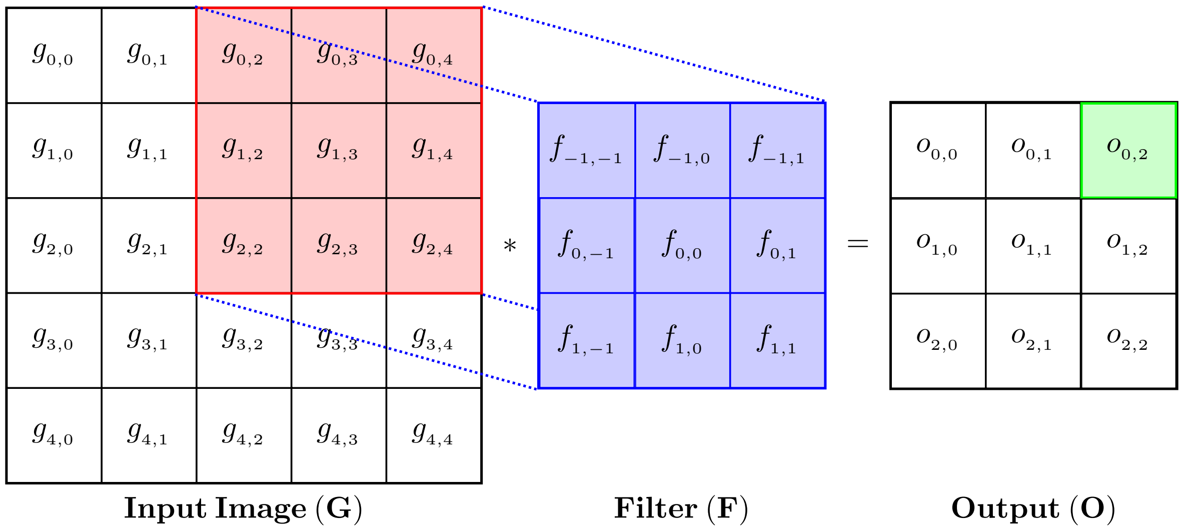

$g_{0,0}$ $g_{0,1}$ $g_{0,2}$ $g_{0,3}$ $g_{0,4}$
$g_{1,0}$ $g_{1,1}$ $g_{1,2}$ $g_{1,3}$ $g_{1,4}$
$g_{2,0}$ $g_{2,1}$ $g_{2,2}$ $g_{2,3}$ $g_{2,4}$
$g_{3,0}$ $g_{3,1}$ $g_{3,2}$ $g_{3,3}$ $g_{3,4}$
$g_{4,0}$ $g_{4,1}$ $g_{4,2}$ $g_{4,3}$ $g_{4,4}$
$*$
$f_{-1,-1}$ $f_{-1,0}$ $f_{-1,1}$
$f_{0,-1}$ $f_{0,0}$ $f_{0,1}$
$f_{1,-1}$ $f_{1,0}$ $f_{1,1}$
$=$
$o_{0,0}$ $o_{0,1}$ $o_{0,2}$
$o_{1,0}$ $o_{1,1}$ $o_{1,2}$
$o_{2,0}$ $o_{2,1}$ $o_{2,2}$
Input Image (G)
Filter (F)
Output (O)

Figure 2.1: Illustration of the convolution operation using a 3x3 filter and a 5x5 input. The result of this convolution on the top right corner of the input is $o_{0,2}$, and can be mathematically formulated as follow: $o_{0,2} = (f_{-1,-1} \times g_{0,2}) + (f_{-1,0} \times g_{0,3}) + (f_{-1,1} \times g_{0,4}) + (f_{0,-1} \times g_{1,2}) + (f_{0,0} \times g_{1,3}) + (f_{0,1} \times g_{1,4}) + (f_{1,-1} \times g_{2,2}) + (f_{1,0} \times g_{2,3}) + (f_{1,1} \times g_{2,4})$.

From Equation 2.1 and Figure 2.1, it can be observed that a convolution is a many-to-one operation. In other words, the values in the input window are multiplied by the filter weights and then summed to return a single output value.

In Figure 2.1, the filter is applied only on the top-right corner of the input, returning a single output value. Repeating this operation by sliding the filter across all coordinates (x, y) in the input will return an output grid, where each value $o_{x,y}$ is the result of the convolution between the filter and the input window whose center coordinates are (x, y). This process is a convolutional layer in its simplest form.

So far, the input has been assumed to be 2D. However, most of the time it will be 3D. The third dimension is usually referred to as the "channels". Therefore, a 2D input can be seen as a 3D input with only 1 channel.

The convolution operations in the case of 2D or 3D inputs do not differ much, with the exception that the number of channels in the filter and the input must be equal. Moreover, the summation and elementwise products will be done over all channels of the input window, which means that, regardless of the number of channels in the input, the number of channels in the output will always be one.

It is usually desirable to have an output with more than one channel. Since any given filter compresses the output channels to one, the key idea is to use as many filters as there should be output channels. Therefore, each filter can be associated with precisely one output channel, while still considering the entirety of the input channels.

In summary, a convolutional layer is a function. Its input can be a 2D or 3D grid of values, whose third dimension is commonly referred to as the number of channels. The function works by applying a series of convolutions between the filter and all possible equal size windows in the input, and its parameters correspond to the values of the filter. The first two dimensions of the output will be similar to those of the input (more on that in the next sections), while the number of channels is controlled by the number of filters used.

2.1.2 Padding

One noticeable side effect of convolutions is the reduction of output dimensions. As illustrated in Figure 2.1, G is 5x5, but O is only 3x3. This is because all windows whose center coordinates are within a certain range of input borders cannot be used; otherwise, the convolution would have multiplications that are out-of-bounds, meaning that a filter value would be multiplied by a non-existing input value.

The number of unusable windows in an input will vary according to the dimension of the filter. More precisely, the output dimensions will be

$$w_O = w_G - w_F + 1, \tag{2.2}$$

where w denotes the width. The same equation applies for the height (h). Therefore, the wider a filter is, the larger the reduction in output size will be.

In many cases, it might be useful to preserve the input shape when applying a convolution, which is done with padding. The two most common padding strategies are "valid" padding and "same" padding. The first one means that no padding will be used, and only the valid regions in the input will be considered, leading to a reduction in the size of the output. The second method, same padding, refers to the action of adding extra values around the input, such that the shapes of the input and output are equal.

Different strategies can be used to decide which values to employ for padding. In most cases, a value of 0 is used, but there also exist many other methods such as reflective padding that can be more appropriate based on the situation. When using padding, the equation for the output dimensions becomes

$$w_O = w_G - w_F + 2P_h + 1, \tag{2.3}$$

where P_h represents the horizontal padding. The same equation applies for the height, replacing P_h by P_v, the vertical padding.

2.1.3 Strides

Earlier, it was mentioned that a convolutional layer computes many convolutions by sliding a filter across all possible coordinates of a given input. However, it is sometimes more efficient and barely penalizing to skip some of them, especially in the case of large size inputs. This is controlled by the stride, a parameter of the convolutional layer.

More precisely, the stride is an integer value greater or equal to one, and corresponds to the number of coordinates to skip every time the filter is slid through the input. So far, all the examples above have assumed a stride of 1, meaning that all patches in the input will be considered. On the other hand, a stride of 2 means that the filter will be applied to every other coordinate in the input, skipping one between each slide. Just like all the other parameters mentioned so far, the stride has a horizontal and vertical component that can have different values. Combined with padding, the equation for computing the output dimensions when using a stride becomes

$$w_O = \frac{w_G - w_F + 2P_h}{S_h} + 1, \tag{2.4}$$

where S_h represents the horizontal stride. The same equation applies for the height, replacing P_h and S_h by P_v and S_v, respectively the vertical padding and stride.

2.1.4 Dilated Convolutions

Dilated convolutions are a variation of regular convolutions with enlarged receptive field. They are often used because they facilitate the detection of long-range dependencies in images, a key feature in inpainting, while also reducing the computational cost. This process is controlled by the dilation rate, a parameter of the convolutional layer that affects how filter values are multiplied with the input.

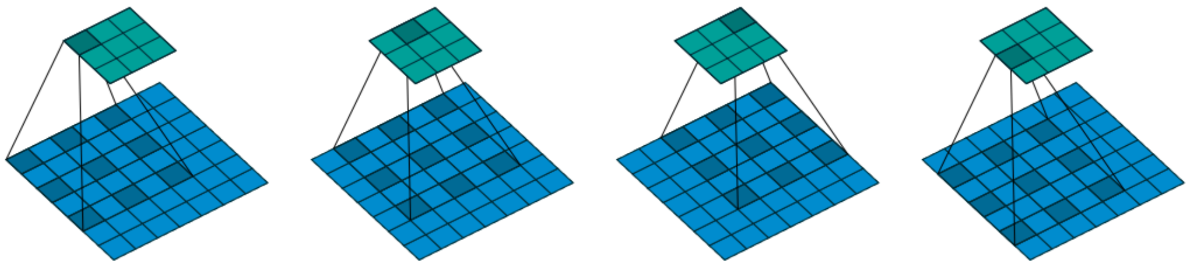


Figure 2.2: Illustration of dilated convolutions, from [6], Figure 5.1.

During a traditional convolution, each value of the filter is mapped to a value of the input, by which it is then multiplied. The mapping rule is simply "both values should have the same relative position with respect to their center coordinates". For example, let's consider only the horizontal axis (the same logic applies to the vertical axis). In this case, if a filter value is at v steps to the right of the filter center, then the input value by which it is multiplied should also be at v steps to the right of the input center. For dilated convolutions, however, if a filter value is at v steps to the right of the filter center, then the input value by which it is multiplied should be at $v \times D_h$ steps to the right of the input center.

In summary, a dilated convolution works just like a regular convolution, with the exception that the filter is not applied to adjacent values in the input. Instead, the dilation rate, an integer value greater than one (one corresponds to a vanilla convolution), controls the number of spaces to insert between kernel values before multiplying it with the input window. Just like regular convolutions, dilated convolutions can have padding and strides. In this case, the equation for the output dimensions becomes

$$w_O = \frac{w_G + 2P_h - [w_F + (w_F - 1)(D_h - 1)]}{S_h} + 1, \tag{2.5}$$

where D_h represents the horizontal dilation rate. The same equation applies for the height, replacing the horizontal components of P, S, and D by their vertical components.

2.1.5 Transposed convolutional layers

Convolutional layers will always produce an output whose width and height are smaller or equal to those of the input. There are many instances where the inverse effect can be desirable. For example, in image inpainting, many regular convolutional layers (RCLs) can be chained and applied to the input, reducing its size. However, the final output is the completed image, and thus, it should have the same shape as the initial input. It is therefore crucial to be able to upsample the input back to its original shape. One way to achieve this is by using some interpolation method to manually increase the dimensions. However, interpolations use a fixed set of parameters, meaning that, unlike convolutional layers, they cannot learn to adapt themselves based on the task at hand. Another option is to use transposed convolutional layers (TCLs), which produce an effect similar to interpolation, but with learnable parameters.

In essence, a TCL is an inverted RCL, which means that it expands the size of the input, as opposed to reducing it. They both have the same parameters, but their effect is, again, inverted. Because they are essentially the same operations, but reversed, the equation for the output dimensions of a TCL can be obtained by inverting w_O and w_G in Equation 2.4, and isolating w_O.

$$\begin{aligned} w_G &= \frac{w_O - w_F + 2P_h}{S_h} + 1 \\ \iff \quad w_O &= (w_G - 1) \times S_h - 2P_h + w_F \end{aligned} \tag{2.6}$$

An intuitive way to compute a TCL is by emulating it with a RCL. First, considering I, a $n \times n$ input and F, a $m \times m$ filter, Equation 2.6 can be used to obtain the expected output size of the TCL. Equation 2.4 can then be used to derive the input size (1) of a RCL with equivalent filter and output sizes. Because the TCL is emulated with a RCL and because the output is supposed to be larger than the input in terms of height and width, the computed input size (1) will always be greater than the actual input size n, meaning that it will be necessary to inflate the input, spacing its values with 0's. This third step is the hardest one, since the 0's need to be inserted in such a way that preserves the connectivity pattern. For example, if the top-left pixel in the input is only connected to the top-left pixel in the output for the RCL, it should be the same for the TCL. Once the input is correctly inflated, the final step is to apply the RCL as usual.

2.1.6 Pooling

The computational complexity of a convolutional layer is directly proportional to the input size. Sometimes, only a small portion of the input is necessary or even beneficial to the convolutional layer. In this case, it is highly desirable to quickly reduce the input size, while preserving only the relevant information. For example, depending on what the convolutional layer is trying to learn, the clarity of edges might be more important than a high-resolution image, which might decrease the thickness of edges. A popular way to achieve this goal is with pooling layers.

A pooling layer works by gathering all the values inside a window, whose size is determined by the pooling window size. The way values are gathered inside a given window will vary according to the strategy. The most common ones are max pooling and average pooling. For max pooling, the maximum value is retained, while for average pooling, the average is computed. All the parameters of a regular convolutional layer can also be applied to a pooling layer.

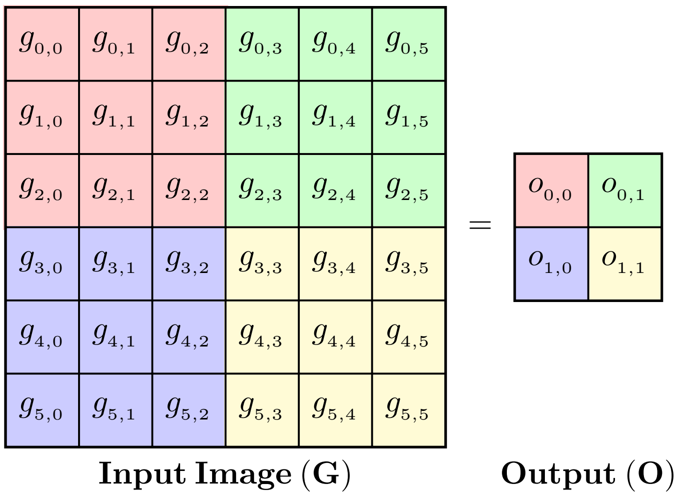

$g_{0,0}$ $g_{0,1}$ $g_{0,2}$ $g_{0,3}$ $g_{0,4}$ $g_{0,5}$
$g_{1,0}$ $g_{1,1}$ $g_{1,2}$ $g_{1,3}$ $g_{1,4}$ $g_{1,5}$
$g_{2,0}$ $g_{2,1}$ $g_{2,2}$ $g_{2,3}$ $g_{2,4}$ $g_{2,5}$
$g_{3,0}$ $g_{3,1}$ $g_{3,2}$ $g_{3,3}$ $g_{3,4}$ $g_{3,5}$
$g_{4,0}$ $g_{4,1}$ $g_{4,2}$ $g_{4,3}$ $g_{4,4}$ $g_{4,5}$
$g_{5,0}$ $g_{5,1}$ $g_{5,2}$ $g_{5,3}$ $g_{5,4}$ $g_{5,5}$
=
$o_{0,0}$ $o_{0,1}$
$o_{1,0}$ $o_{1,1}$
Input Image (G)
Output (O)

Figure 2.3: Illustration of a pooling layer, with 0-padding and a stride of 3. For example, $o_{0,0} = \max(g_{0,0}, g_{0,1}, g_{0,2}, g_{1,0}, g_{1,1}, g_{1,2}, g_{2,0}, g_{2,1}, g_{2,2})$ if a max pooling strategy is used, and $o_{0,0} = (g_{0,0} + g_{0,1} + g_{0,2} + g_{1,0} + g_{1,1} + g_{1,2} + g_{2,0} + g_{2,1} + g_{2,2})/9$ if an average pooling strategy is used.

2.1.7 Activation functions

So far, we have defined convolutional layers with multiplications and additions, two basic arithmetic operations. With only these operations, a convolutional layer would be a linear operation, which is not desirable because CNNs are often used to model phenomena that do not necessarily have a linear behavior. Therefore, to ensure that convolutional layers can also reproduce non-linear patterns, activation functions are utilized. These are non-linear functions, applied to the output of a convolutional layer. Therefore, convolutional layers can be non-linear if coupled with an activation function, or linear if not. Figure 2.4 displays some of the most common activation functions used, but there exist many more.

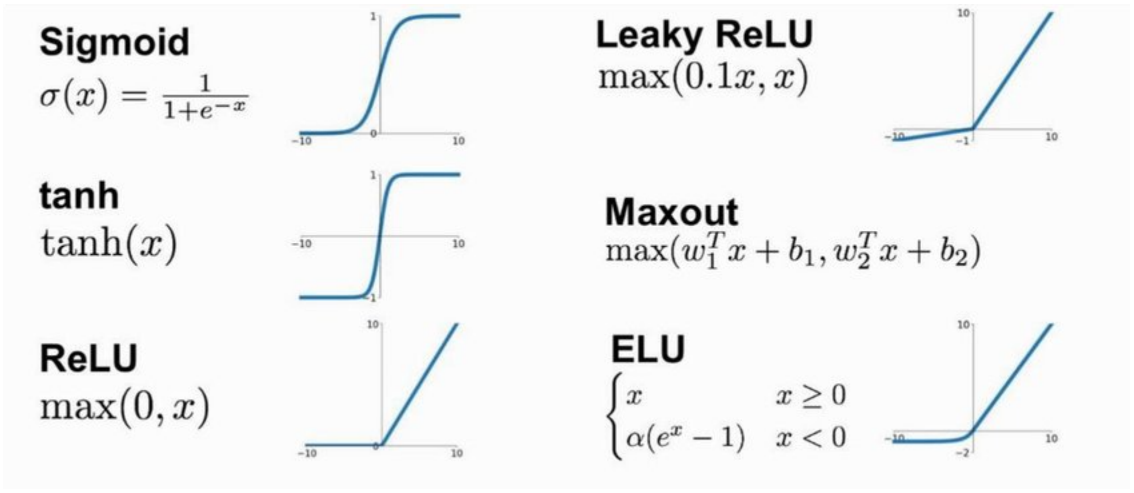


Figure 2.4: Most common activation functions in deep learning, from [14], figure 3.

2.1.8 Softmax

Neural networks are used for predictive modelling. In their simplest form, they predict the outcome of a binary event, such as "will it rain (0) or not (1) tomorrow?" However, there are many scenarios for which there are more than only two possibilities. For example, in inpainting, the output is a map containing pixel value predictions at every location in the image. Assuming that pixel values can range from 0 to 255, this means that there are 256 possibilities, making it a multi-class problem. Another simpler example is image classification. In this case, the objective could be to classify an image into different categories such as cat, dog, or plane.

Unless using an activation function that outputs integer values, a convolutional layer will output an array or a matrix of float values. However, it can be useful to normalize these values into probabilities for multi-class problems. Indeed, float values can be hard to interpret, especially small and high values, whereas probabilities have an intuitive meaning and are easily comparable.

Softmax is a non-linear function used to normalize a vector into a probability distribution. More precisely, let's define a vector $\vec{v}$ of dimension $n \times 1$. The softmax function will then be

$$\sigma(\vec{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{n} \exp(v_j)}, \tag{2.7}$$

where i and j are respectively the i-th and j-th values of $\vec{v}$. From the equation above, it can be observed that $\sum_{i=1}^{n} \sigma(\vec{v})_i = 1$ and $0 \leq \sigma(\vec{v})_i \leq 1 \; \forall i$, so that $\sigma(\vec{v})$ is a probability distribution.

2.1.9 Training and Gradient Descent

The power of CNNs resides in their ability to learn and adapt to different tasks. Just like simple linear functions, whose parameters can be derived with mean squared error such that they best fit the underlying data, CNN parameters are optimized in order to return the best possible results. Considering that CNNs often have millions of parameters and are complex non-linear functions, it is usually impossible to find a closed-form expression for the optimal parameters; instead, gradient descent is utilized.

CNNs can be in one of two states: training or inference. During training, the correct result is known, and the parameters of the model are optimized in a way that minimizes the difference between the true result and model output. To accomplish this, the first step is to define a loss function that adequately captures the goal of the task. When considering an image restoration task for example, it could be a mean squared error between pixel values in the masked regions of the output image and ground truth. In this case, the desired output should be as similar as possible to the original image, which is achieved by reaching the minimum of the loss function. Therefore, the second step is to gradually update the parameters, which, for CNNs, are filter values, in order to minimize the loss function.

Since the domain of the loss function is m-dimensional, with m usually greater than one million, finding the minimum is not trivial, which is why the parameters are updated by small increments over many steps. This whole process is called gradient descent.

The loss function variables are the input values, while its parameters are the filter values. Gradient descent works by computing the derivative of the loss function with respect to every individual parameter and subtracting it from the current values of the parameters. Let L be the loss function, θ the set of parameters, and m the size of θ. Then, the gradient will be a vector ∇g of size $m \times 1$, where $(\nabla g)_i = dL/d\theta_i \, \forall \theta_i \in \theta$. In this case, the parameters are updated with the following formula:

$$\theta_{n+1} = \theta_n - \alpha \nabla g_{n+1}, \tag{2.8}$$

where n and $n+1$ respectively denote the previous and current steps, ∇g_{n+1} is the gradient computed at the $n + 1$ step, and α is the learning rate. The learning rate is a scaling hyperparameter that controls the speed at which parameters are updated.

Once the model is finished training, the parameters are frozen at their current values, and the state switches to inference. At this point, the true answer is not known, and the model is used for prediction.

2.1.10 Common architectures in Image Inpainting

The main components of convolutional neural networks have been presented so far, without formally defining CNNs themselves. In simple terms, a CNN is a combination of all the elements presented above, built by chaining multiple regular and transposed convolutional layers, pooling layers, activation functions, and softmax units. More formally, it is a function that takes a 3D matrix as input, and returns another matrix, vector, or even scalar, depending on how it is assembled.

The architecture of a CNN is usually heavily influenced by the nature of the task. In inpainting, common architectures usually feature an encoder and a decoder. An encoder mainly comprises regular convolutional layers as well as pooling layers. Its purpose is to transform the initial input into its latent representation by extracting the relevant information to the model. This is done by compressing the width and height of the initial image, while expanding the number of channels. A decoder, on the other hand, is usually built with a mixture of transposed and regular convolutional layers. It transforms the input from its latent representation back to its original format, in this case an RGB image, by expanding its width and height and compressing its channels. The rationale behind this encoder-decoder strategy is that the latent space might be helpful to learn data features and highlight similarities in data, which can then be used by the model during the decoding stage.

It is common to see multiple encoder-decoder architectures chained together in image inpainting. Indeed, one pair can work on obtaining a coarse, blurry result, while another

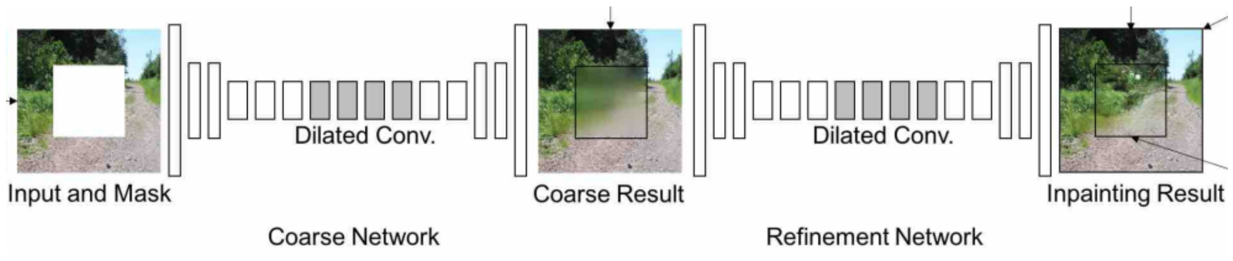


Figure 2.5: Example of coarse-to-fine architecture in image inpainting, from [34], figure 2. The first encoder-decoder (denoted the "coarse network") is used to get a coarse result, a blurry approximation of what the pixel values should be inside the hole region. The second pair of encoder-decoder, represented as the "refinement network", uses the coarse approximation to get a final, clear result.

pair works on refining that result into a clearer output. This way, the second encoder-decoder has access to more information than simply black and white pixels. For example, in Figure 2.5, even though the coarse result is extremely blurry, it is easy to see that the color green is prevalent in the top left corner of the image, which tells the refinement network that this area might be similar to the trees in the top-left region of the image. Such architectures are commonly referred to as "coarse-to-fine".

2.2 Residual learning

Intricate problems like inpainting usually have a more complex underlying distribution. Consequently, they require more parameters to be adequately modeled, causing the network to be deeper. Such models often struggle with vanishing or exploding gradients. As explained in Section 2.1.9, training a neural network involves computing the derivative of the loss function with respect to each parameter. For deeper models, these derivatives will have more terms and more multiplications. For example, let's consider a simple model with n layers, where, at each layer, input value x is multiplied by parameter θ_i, for $1 \leq i \leq n$. The derivative with respect to θ_i will then be $(\prod_{1 \leq j \leq n, j \neq i} \theta_j) \times x$, and it is easy to see that, for a large n, the result will be very large or very small if $\theta_i > 1 \forall i$ or $\theta_i < 1 \forall i$, respectively. The former is called an exploding gradient, and the latter, a vanishing gradient.

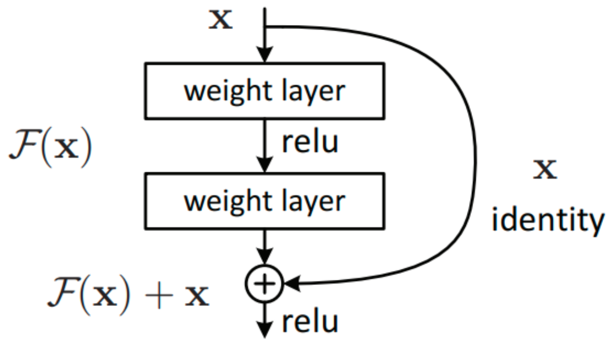


Figure 2.6: Building block of residual learning, from [10], Figure 2.

Residual learning, proposed by He et al. [10], is a framework that stabilizes the training of deep neural networks. It works by concatenating the output of a layer with the output of a deeper layer, and is implemented by stacking one or more residual blocks. Every block features two layers, each activated with ReLU, and the input of the first is concatenated to the output of the second, before its activation function. This concatenation, commonly referred to as a "skip connection", emulates an identity function by creating a "shortcut" that allows the input to travel deeper into the network while retaining its original values.

Being able to simulate an identity function is crucial for deep models. Indeed, if no further learning can be done before reaching the last layer, the model should be able to copy the current output and ignore the next layers. However, this can be hard to achieve through straightforward parameter optimization, which is why skip connections are so powerful. Without them, the input is bound to be modified until the very last layer, even if its optimal state has already been reached, which can only hurt performance. They also help to avoid exploding and vanishing gradients, since the identity function does not use any additional parameters, and consequently, the expression for the derivative will be composed of fewer terms and multiplications.

2.3 Mask generation

Mask generation is a central component of image inpainting since it influences how parameters are optimized during training. Indeed, the model learns by filling regions hidden by the mask. Therefore, if the mask is biased in any way, it could negatively impact how the network is trained. For example, if the hidden part of the image always exclusively contains trees, then the model will learn how to complete trees, but it will likely fail if asked to complete an airplane at inference time. A sound mask generation strategy should be adjustable and computationally efficient, while also producing masks that are similar to real user inputs and diverse to avoid over-fitting [35].

2.3.1 Square masks

Square masks are the oldest and simplest variety. As their name suggests, they are used to randomly hide square portions of the image. While being efficient to produce, square masks do not mimic real user input. Moreover, if the user were to be limited to squares only for designating the area to complete, the lack of freedom would likely lead to obfuscating a larger area than necessary. In this case, important information that could have, otherwise, been available to the model for learning would be hidden, thus hindering training. Also, pixels closer to the center of a square are far away from the borders, meaning that they will not have a lot of valid context available. Lastly, the diversity is limited to only one shape.

Natural shapes other than squares, such as circles and ellipses, can also be used. Although they are more flexible than square masks, they still share many of their flaws.

2.3.2 Free-form masks

Free-form masks aim to be a more plausible representation of real inputs. They are created by randomly simulating brush strokes, which is likely how a user would indicate the regions to fill, since brush strokes have a high precision and are customizable. Free-form masks are diverse, plausible, and highly adjustable, but they come at a greater computational cost. Algorithm 1 describes the generation process of free-form masks.

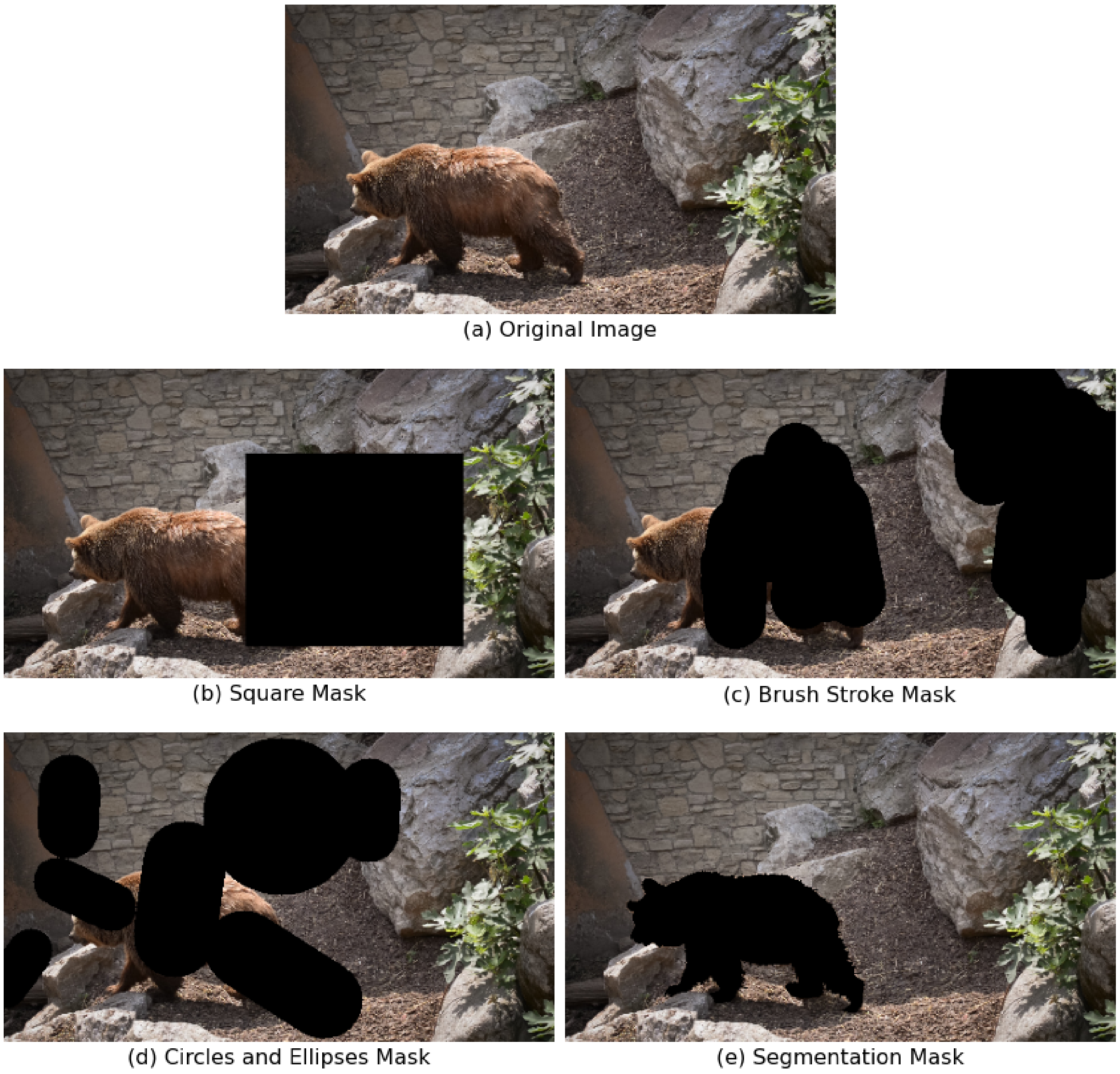


Figure 2.7: Illustration of different mask generation strategies. Image and segmentation mask were taken from DAVIS 2016 dataset [22].

Algorithm 1: Algorithm for brush stroke masks. Hyper-parameters maxVertex, maxLength, maxBrush-Width, and maxAngle control the size of the generated mask. Taken from [35], Algorithm 1.

```
mask = zeros(imageHeight, imageWidth);
numVertex = random.uniform(maxVertex);
startX = random.uniform(imageWidth);
startY = random.uniform(imageHeight);
brushWidth = random.uniform(maxBrushWidth);
for $i = 0$ to $numVertex$ do
    angle = random.uniform(maxAngle);
    if (i % 2 == 0) then
        angle = 2 * pi - angle;
    end
    length = random.uniform(maxLength);
    Draw line from point (startX, startY) with angle,
      length and brushWidth as line width;
    startX = startX + length * sin(angle);
    startY = startY + length * cos(angle);
    Draw a circle at point (startX, startY) with radius
      as half of brushWidth;
end
```

2.3.3 Segmentation masks

A very common use case for inpainting is the removal of an undesirable object from the background. Although free-form masks imitate well the shape of hidden areas to complete that would result from this task, they do not target a specific region; instead, they randomly draw brush strokes across the image, often spanning multiple different objects. Segmentation masks, on the other hand, precisely designate an area whose pixels all belong to the same object, which simulates well the case of object removal. They are, however, hard to obtain, since they require manual labelling. Moreover, they are much more precise than a real user input. For example, the hair of the bear on Figure 2.7(e) shows the granularity and precision of the mask, which is unlikely to be the case for a user input. Nonetheless, their usage can stabilize training and return a better evaluation of the model's performance.

2.3.4 Parameters

Choosing a suitable mask generation strategy is a vital step in designing a good inpainting model, and the shape of the mask is not the only factor to consider. Indeed, the percentage of the image to hide should be selected carefully and right for the type of model used. For example, larger masks are often more appropriate for pluralistic inpainting since the region to complete is not constrained as much by the valid context [30]. Also, the masked area percentage does not have to be constant over time; instead, a valid strategy is to initially have the model focus on accurately learning details with small masks, and then gradually switching to a more global understanding of the image with larger masks [29].

2.4 Generative adversarial networks

One prerequisite for completing images, in inpainting, is the ability to generate new content. Additionally, this content should be plausible for the final result to look realistic. However, plausibility is a relative concept specific to each task. In inpainting, for example, it means that the generated areas should have colors, structure, and texture similar to the uncorrupted parts of the image. One possible approach to generalizing this concept in deep learning is to consider generated samples plausible only if they are drawn from the same distribution as the training examples. In computer vision, the data distribution corresponds to the set of images on which the model is trained, and can represent, for example, natural landscapes, human faces, or indoor architecture. Unlike for a Gaussian distribution, randomly generating a sample that belongs to the set of natural landscapes is not trivial. Indeed, this type of distribution is not well defined, and there is no closed form formula to express it, making simple generating methods like inverse transform sampling impossible to apply.

In 2014, Goodfellow et al. [8] proposed a new framework called generative adversarial networks, designed to generate random samples from complex data distributions using neural networks. In essence, it is built with two networks competing against each other in a minimax type of game. The generator, G, captures the data distribution, while the discriminator, D, tries to detect if a given sample is drawn from the training data or produced by G. This whole system is optimized with the following equation:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))], \tag{2.9}$$

where z is a random n-dimensional data point sampled from a known distribution, p_z, and p_{data} is the training data distribution, or, in other words, the set of all training examples.

D outputs a score indicating the likelihood that a sample originates from the training data, where a high score corresponds to a high likelihood. At the beginning, when G is still a bad estimate of p_{data}, D can easily distinguish a training example from the output of G, given that the latter is mostly random data. Therefore, $\log(1 - D(G(z))$ tends to saturate, which is why, in practice, $\min_G\{\log(1 - D(G(z))\}$ is usually replaced with $\max_G\{\log(D(G(z))\}$, which is equivalent and produces more stable gradients. Also, to reduce the computational cost, and avoid overfitting, G is optimized every k steps, while D is optimized at every step. Algorithm 2 describes the training process of generative adversarial networks.

Algorithm 2: Algorithm for training adversarial neural networks with stochastic gradient descent. k is an hyper-parameter that controls the number of steps between each optimization of the generator. α and β are learning rates, two other hyper-parameters. Taken from [8], Algorithm 1.

```
1  for $n = 0$ number of training iterations do
2      for $k$ steps do
3          ➢ Draw random sample $z$ from distribution $p_z$
4          ➢ Draw random training example $x$ from $p_{data}$
5          ➢ Update discriminator using gradient descent:
6          $\theta_{D_{k(n+1)+1}} = \theta_{D_{k(n+1)}} + \alpha\nabla_{\theta_D}[\log D(x) + \log(1 - D(G(z))]$
7      end
8      ➢ Draw random sample $z$ from distribution $p_z$
9      ➢ Update generator using gradient descent:
10     $\theta_{G_{n+1}} = \theta_{G_n} - \beta\nabla_{\theta_G}\log(1 - D(G(z)))$
11 end
```

Note: any other standard gradient learning rule can be used to update θ_D and θ_G. θ_{D_t} and θ_{G_t} respectively denote the parameters of D and G at step t. The update for G can be equivalently replaced with $\theta_{G_{n+1}} = \theta_{G_n} + \beta\nabla_{\theta_G}\log(D(G(z)))$ as mentioned earlier.

In summary, generative adversarial networks employ neural networks to generate data samples from complex distributions. Indeed, neural networks are not limited and can be expanded to millions of parameters, which provides a good estimation for these complex distributions. Moreover, because they are trainable, they can be adapted to a multitude of different tasks.

2.4.1 PatchGAN

As described above, D outputs a scalar that represents the likelihood of the input belonging to the training samples, p_{data}. In inpainting, though, because the image has two components, the masked region and the valid context, it is crucial to have the flexibility to target different areas, and not just consider the input as a whole. In 2017, Iizuka et al. [12] proposed a network with two discriminators, one that looks at the image in its entirety, while the other only considers the region to fill, enforcing global and local consistency simultaneously. While this works reasonably well and allows to specifically target the area to complete, it is only applicable to square masks. Indeed, it is straightforward to carve out square shaped regions, and feed them to the second discriminator. However, in all the other cases where the mask has a free-form shape, segmenting the area to inpaint will result in a non-square form, which is an invalid input shape for CNNs, usually utilized in computer vision to implement the discriminator.

In 2019, Yu et al. proposed SN-PatchGAN [35], a modified version of GANs that allows attending to precise regions in the image, and works for masks of all shapes. Instead of returning a scalar, the discriminator of SN-PatchGAN, S, is built to output a $h \times w \times c$ 3-D map. The receptive field of each location (i, j, k) in this map is a spatial area of the original input, and may or may not, depending on the stride and dilation rate used, overlap with regions covered by other locations. Each output value represents the likelihood that the area it covers is part of a real training example. In this case, Equation 2.9 has to be applied $h * w * c$ times, once at every location, and the optimization rule then becomes

$$\begin{aligned} \min_G \max_S V(S, G) = \mathbb{E}_{x \sim p_{data}(x)}\Bigg[\sum_{(i,j,k)\in A} \log S_{i,j,k}(x) \Bigg] \\ + \mathbb{E}_{z \sim p_z(z)}\Bigg[\sum_{(i,j,k)\in A} \log\left(1 - S_{i,j,k}(G(z))\right) \Bigg], \end{aligned} \tag{2.10}$$

where A is the set of all possible locations in the $h \times w \times c$ output map, and $S_{i,j,k}(x)$ is the score at location (i, j, k) resulting from applying S on input x.

One final consideration is the mask. If S needs to consider the corrupted regions with greater attention than the valid context, then the number of masked pixels in the receptive field of each output value can be counted and used to weigh $\log S_{i,j,k}(x)$ and $\log\left(1 - S_{i,j,k}(G(z)\right)$ in Equation 2.10. This way, regions with no hidden pixels are multiplied by a weight of 0, and do not affect the training of S.

2.5 Contextual attention

Patch-based methods rely on the assumption that most, if not all, of the missing content in a corrupted image is contained in the known regions of that same image. While this is a reasonable hypothesis for images with repetitive patterns, it is often not true for complex structures. On the other hand, CNNs, and particularly GANs, are great at hallucinating new content for most type of images. Nonetheless, they do not excel in detecting long-range dependencies. Contextual attention [34] aims to solve this flaw by taking the idea behind patch-based methods, and integrating it to CNNs to improve their ability to borrow information from distant spatial locations.

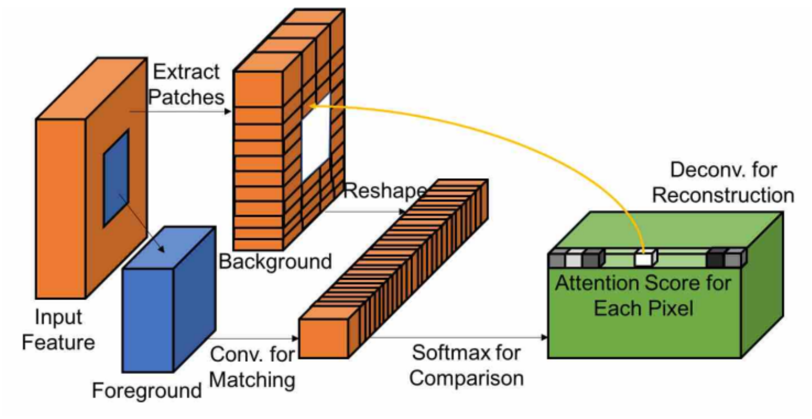


Figure 2.8: Illustration of contextual attention, from [34], Figure 3.

In inpainting, there are two parts to the input: the foreground, the area to inpaint, and the background, the valid context. With contextual attention, the goal is to find similarities between foreground and background areas, which is done using cosine similary:

$$s_{x,y,x',y'} = \langle \frac{f_{x,y}}{\|f_{x,y}\|}, \frac{b_{x',y'}}{\|b_{x',y'}\|} \rangle, \tag{2.11}$$

where $s_{x,y,x',y'}$ is the similarity score between foreground patch f, centered at coordinates (x, y), and background patch b, centered at coordinates (x', y').

Let's now define the whole process more formally. Let I be an $n \times m \times c$ input. The first step is to extract patches of size $p \times p \times c$ from I. The total number of patches that can be extracted is given by Equation 2.5. Indeed, multiplying w_O with h_O returns the number of output values from a convolutional layer, all channels included. Because each output value is associated with a unique input window, and because all input windows are considered, $w_O \times h_O$ also returns the number of patches that can be extracted from I, which will be denoted by T.

The next step is to compute similarities between patches and foreground values. This can be done with a regular convolutional layer, using the extracted patches as filters, and setting those that belong to the foreground to 0. This way, only similarity scores with background patches will be greater than 0. Since there are T patches, each of size $p \times p \times c$, the returned output shape will be $w_O \times h_O \times T$, where $O_{x,y,z}$ corresponds to the similarity score between the input window centered at (x, y) and patch z. A softmax unit is then used to normalize these similarity scores into probabilities called attention scores.

The final output is a sum of the extracted patches, weighted by their attention scores, for every location (x, y). Since the foreground region can be of any shape, it is hard to isolate foreground values from background values, which is why the weighted sum is computed for all pixels in the input. However, only foreground values are retained, while background values are set back to their original input value.

One obvious limitation of this module is the lack of reliable information for the foreground area. Initially, values in this region are all set to 0, which makes it impossible to compute similarity scores. This is why contextual attention is usually computed deeper into the model, when the corrupted area has been filled with approximate values. Coarse-to-fine architectures are especially useful in this context, since they produce an early coarse result, which can be used as an approximation of the foreground region.

Contextual attention is also computationally expensive, which is why it is often used at smaller scales to reduce the number of patches to compute similarities with. In this case, the mask is downsized to the shape of the current input using an interpolation method or average pooling to determine what regions belong to the foreground and background at this scale.

2.6 Gated convolutions

As seen in Section 2.1.1, convolutional layers work by applying convolutions all over the input. However, in inpainting, many pixels are invalid, and should not be treated like the valid ones. Gated convolutions implement this idea by weighing input values differently such that only uncorrupted pixels affect the output value. This is accomplished with gating weights that are initially set to the mask values. There exists two types of gated convolutions: soft gating [35] and hard gating [19]. In essence, hard gating convolutions either consider a pixel valid (1), or not (0), whereas soft gating also allows for partially valid pixels $\big(]0,1[\big)$.

2.6.1 Hard gating

Let's define X, the feature values of the current input window, G, its corresponding gating weights, where 1's and 0's respectively indicate valid and invalid pixels, F, the filter of the convolution, and o, the resulting output value. X, G, and F are all $m \times n$ matrices, with (0,0) as their center coordinates. Negative filter indices are allowed. Then, a hard-gating convolution can be expressed as:

$$o = \begin{cases} \frac{1}{\text{sum}(G)} \sum_{i=-k_h}^{k_h} \sum_{j=-k_v}^{k_v} F_{i,j} \times (X_{i,j} \times G_{i,j}), & \text{if sum}(G) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.12}$$

Just like for regular convolutions, $k_h = \frac{m-1}{2}$, and $k_v = \frac{n-1}{2}$ for m and n odd. If m and n were even, k_h and k_v would respectively become $\frac{m}{2}$ and $\frac{n}{2}$, and the lower bounds of the summations would instead be $-k_h + 1$ and $-k_v + 1$.

The main difference with a regular convolution is that pixel values are not only multiplied by filter weights, but also by a gating weight; therefore, the output value will not be a function of corrupted pixels given that they are set to 0. Also, since different windows can have different gating weights, the returned value is an average instead of a sum; otherwise, windows with fewer valid pixels would likely have a smaller output value. After each convolution, the gating weights are updated with the following equation:

$$G' = \begin{cases} 1, & \text{if sum}(G) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.13}$$

This means that if an output value had at least one valid pixel used in its computation, then it will be marked as valid for the next layer. Eventually, if the model is deep enough, all the pixels will be marked as valid. In summary, a hard gating convolution, along with its update rule, pushes the model to completely ignore corrupted pixels at first, and then slowly trust them in deeper layers.

2.6.2 Soft gating

Soft gating is built on the same idea as hard gating, but it instead uses learnable parameters to update the gating weights after each convolution, and supports values between 0 and 1 to denote partially valid pixels. More precisely, the convolution is carried out as usual on the input ($batchsize \times height \times width \times channels$), but another convolution, with a different set of filters, also happens in parallel on the gating weights ($batchsize \times height \times width \times channels$). The result of the second convolution is then activated with a sigmoid function, mapping the values to the $[0, 1]$ range, and multiplied with the output of the first convolution. The result of this last multiplication is then passed to the next layer, along with the gating weights resulting from the second convolution, and the whole process is repeated.

With this technique, the model is able to learn a dynamic feature selection mechanism that is specific to each channel and spatial location. The main downside of soft gating is that it doubles the number of parameters to train because of the second convolution, thus leading to a substantial increase in training time.

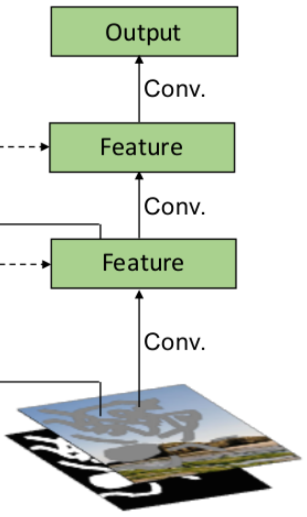


Figure 2.9: Illustration of soft gating convolutions, from [35], Figure 2. The convolution is applied on the input as usual, while an extra convolution is also applied on the gating values, which are then activated by a sigmoid function and multiplied with the feature output.

2.7 Loss functions

Section 2.1.9 describes the training process of neural networks and explains why a good loss function that captures the intricacies of the underlying task is crucial. The following sections will present the most common components of loss functions for inpainting.

2.7.1 $\mathbf{L_1}$ and $\mathbf{L_2}$

L_1 and L_2 are two losses that enforce similarity at the pixel level between the reconstructed and original images. More precisely, L_1 is the sum of pixel-wise absolute differences between ground truth and predicted images

$$L_1(y^{\text{true}}, y^{\text{predicted}}) = \frac{1}{n} \sum_{(i,j,k)\in P} |y^{\text{true}}_{i,j,k} - y^{\text{predicted}}_{i,j,k}|, \tag{2.14}$$

while L_2 is the sum of pixel-wise squared differences between ground truth and predicted images

$$L_2(y^{\text{true}}, y^{\text{predicted}}) = \frac{1}{n} \sum_{(i,j,k)\in P} (y^{\text{true}}_{i,j,k} - y^{\text{predicted}}_{i,j,k})^2, \tag{2.15}$$

where P, of size n, is the set of valid 3-D coordinates for pixels in y_{true} and $y_{predicted}$. Each coordinate is represented as $(i,\ j,\ k)$, respectively the row, column, and RGB channel.

L_1 is usually preferred over L_2 for image reconstruction tasks. Indeed, because L_2 squares the error, it is greatly affected by outliers. Consequently, it overweights regions where the difference of magnitude between pixels is large, usually around the edges, leaving visual artifacts in flat areas [37].

2.7.2 Structural loss

When applied directly on pixel values, L_1 and L_2 aim to replicate colors and texture of the original image to the reconstructed areas. However, depending on the input, they can also be used to enforce structural consistency. For example, gradient fields and edges provide valuable structural information about an image. In 2020, Yang et al. introduced a pyramid structure loss [33], in which gradient fields of the inpainted and ground truth images are given as input to a L_1 loss. They also add a regularization term to consider edges and their immediate surroundings with extra weight. More formally, let's define I, the input image, G, its gradient field, and E, its edge map. G can be computed using sobel filters, while E is obtained with canny edge detector. Then, the loss can be expressed as

$$L_{structural} = \frac{1}{n} \sum_{(i,j,k)\in P} |G^{\text{true}}_{i,j,k} - G^{\text{predicted}}_{i,j,k}| + \frac{\alpha}{\text{sum}(E')} \sum_{(i,j,k)\in P} |G^{\text{true}}_{i,j,k} - G^{\text{predicted}}_{i,j,k}| \times E'_{i,j,k}, \tag{2.16}$$

where P, of size n, is the set of valid 3-D coordinates for pixels in G_{true}, $G_{predicted}$, and E'. α is a hyper-parameter used to control the weight of the regularization term. E' is a dilation of the ground truth edge map obtained by convolving E with a 5×5 standard Gaussian kernel, which is equivalent to blurring the edges, and causes values near edges to be considered as well.

In summary, the pyramid structure loss is the sum of pixel-wise absolute differences between image gradients, where extra weight is added around edges through a regularization term, and is concerned with faithfully reconstructing structural information.

2.7.3 Perceptual loss

So far, presented losses have been concerned with reconstructing an image as similar as possible to the ground truth. However, inpainting is not deterministic, but instead pluralistic. Even though there exist multiple solutions, they should be coherent and consistent with the surrounding context. Section 2.4 presents a first approach to defining these concepts, in which an image is considered plausible if it is drawn from the same data distribution than the training samples. Another popular method to measure and enforce plausibility is the perceptual loss, introduced by Johnson et al. [15].

The perceptual loss uses a pre-trained model, such as VGG-19 [25], to extract high-level features from the ground truth, y^{true}, and the inpainted result, $y^{\text{predicted}}$. It then compares them using a L_1 loss, which penalizes the network if it produces a result with significant feature differences.

$$L_{perceptual} = \frac{1}{\text{n}} \sum_{i\in N} L_1\big(\phi^{(i)}(y^{\text{true}}), \phi^{(i)}(y^{\text{predicted}})\big), \tag{2.17}$$

where ϕ is the pre-trained model, $\phi^{(i)}$ is the activation map of its i-th layer, and N, of size n, is the set of layers to compare.

The intuition behind this loss can be summarized as follows: given that the input is only partially covered, the completed result should mostly contain the same features as the original image. Therefore, if they are significantly different, the reconstructed region is likely not coherent with the rest of the image, and thus, the model should be penalized.

2.7.4 Adversarial loss

Section 2.4 explains why using a generator and discriminator can greatly improve performance in inpainting, and gives the theoretical approach to optimizing the system via Equations 2.9 and 2.10. However, under this setup, it is often hard to converge to the optimal solution, which is why, in practice, the hinge loss is incorporated to stabilize the training. Equation 2.10 then becomes

$$\begin{cases} L_D = \frac{1}{n}\sum_{i,j,k\in A}\left[\max\{0, 1 - D_{i,j,k}(x)\} + \max\{0, 1 + D_{i,j,k}(G(z))\}\right] \\ L_G = \frac{1}{n}\sum_{i,j,k\in A} D_{i,j,k}(G(z)), \end{cases} \tag{2.18}$$

where G is the generator and z is a random n-dimensional data point sampled from a known distribution. D is the discriminator of SN-PatchGAN and returns a $h \times w \times c$ 3-D map. A, of size n, is the set of all possible locations in this $h \times w \times c$ output map, and $D_{i,j,k}(x)$ is the score at location (i, j, k) resulting from applying D on input x, where a high score corresponds to a high likelihood of the input being sampled from the training examples. The whole system can be optimized by minimizing L_D with respect to θ_D, the parameters of D, and maximizing L_G with respect to θ_G, the parameters of G.

Another common practice to further stabilize the training of D is to use the feature matching loss [31]. Proposed by Wang et al. , this loss enforces consistency of discriminator features. More precisely, it computes the L_1 difference between D's intermediate activation layers when given the ground truth of an input image, x, and its reconstructed counterpart $G(z)$.

$$L_{FM} = \frac{1}{\mathrm{T}}\sum_{i=1}^{\mathrm{T}} L_1\big(D^{(i)}(x), D^{(i)}(G(z))\big), \tag{2.19}$$

where $D^{(i)}$ is the activation map of the i-th layer of the discriminator, and T is the total number of layers. L_{FM} is minimized with respect to θ_G, the parameters of G.

The objective of this loss is to stabilize the training by pushing discriminator features of real and fake inputs close to each other in the latent space. Indeed, as explained in Section 2.4, G is only optimized every k epochs to reduce computational cost and avoid overfitting. Consequently, the discriminator rapidly gains superiority over the generator in the earlier iterations. If G is not able to catch up, the system will likely not converge to the optimal solution. The feature matching loss helps G to improve faster by giving it insights as to what features the discriminator is looking for in an input to categorize it as real, which stabilizes the whole system.

2.8 Evaluation Metrics

As described in the introduction, evaluation of inpainting results is hard because there exists more than one solution, and the quality of each is relative to the task considered. The next sections will summarize a few common ways to evaluate inpainting results.

2.8.1 $\mathbf{L_1}$

As described in Section 2.7.1, L_1 measures the resemblance of the completed result to the original image. The equation to compute this metric is given by Equation 2.14. L_1 is an appropriate score for problems like image reconstruction, where the model should faithfully reproduce pixels in the corrupted regions. However, it is not useful for pluralistic models, or object removal, since the completed area is expected to be different than what it was originally.

2.8.2 Peak signal-to-noise-ratio

The peak signal-to-noise-ratio (PSNR) is a metric based on the L_2 loss (2.15), and is used in computer vision to assess the similarity between two images. In essence, it is a ratio between the maximum possible intensity of a signal, MAX_I, and the noise affecting the exactness of its representation, quantified using L_2. The peak signal-to-noise-ratio is expressed as follows:

$$\mathrm{PSNR}(y^{\mathrm{true}}, y^{\mathrm{predicted}}) = 20 \log_{10} \left(\frac{\mathrm{MAX}_I}{\sqrt{L_2(y^{\mathrm{true}}, y^{\mathrm{predicted}})}} \right), \tag{2.20}$$

where y^{true} and $y^{\mathrm{predicted}}$ are respectively the original image and the inpainted result. The log of the ratio is taken to normalize the results given that many signals have very different ranges, and therefore, maximum intensities.

This ratio can also be seen as the number of times the completed image is better than the worst possible case, normalized by taking the log. For example, for pixels in the range [0,255], the furthest away two images can be from each other is if one is black (all 0's) and the other is white (all 255's). In this case, the L_2 difference will be $255^2 = 65{,}025$, the average squared difference between two pixels, and MAX_I, 255, divided by the square root of 65,025, will return 1, meaning that the reconstructed image is not better than the worst output.

2.8.3 Structural similarity index

L_1 and PSNR are strictly numerical comparisons and do not take into account factors perceived by the human eye, which is what the structural similarity index (SSIM), proposed by Wang et al. [32], aims to do. Indeed, the human perception is largely influenced by three main factors: luminance, contrast, and structure. Therefore, if the completed image shows an abrupt difference in any of these three areas with the original image, it is likely to be noticeable. The SSIM takes as input two images, x and y, and returns a score that reflects how different they are in terms of luminance, contrast, and structure. Each factor is calculated individually, and combined into a final SSIM score, where a high value symbolizes a great similarity.

Luminance in an image is characterized by the average magnitude of pixel values. Indeed, a dark image will have more pixels with value close to 0, whereas for a bright image this value will be 255. Luminance is computed as follows:

$$l(x, y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1}, \tag{2.21}$$

where μ_x and μ_y are the average pixel values in x and y, respectively. c_1 is a small constant to stabilize the division when the denominator is close to zero. It can be shown that the maximum value for the luminance is 1, and is reached only if $\mu_x = \mu_y$, meaning that the two images are of equal brightness or darkness.

Contrast in an image is measured with the standard deviation of pixel values. A large spread around the mean indicates that different regions in the image have very different intensities, creating contrast. The equation for this factor is given by:

$$c(x, y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2}, \tag{2.22}$$

where σ_x and σ_y are the standard deviations of pixel values in x and y, respectively. c_2 is a small constant to stabilize the division when the denominator is close to zero. It can be shown that the maximum value for the contrast is 1, and is reached only if $\sigma_x = \sigma_y$, meaning that the two images contain pixels equally spread around the mean.

Unlike luminance and contrast, structure uses the covariance to compare each pixel individually, as opposed to comparing aggregate statistics. More precisely, the covariance computes the difference in pixel intensity with respect to the image mean at a given location for both x and y, and multiplies them together. This operation is repeated for every location in the images, and combined into the structure score, $s(x, y)$, to evaluate if the intensity variation is similar within each pair. This is particularly great to assess if two images have matching structural patterns, since change of intensity usually happens around the edges. The equation for structure is expressed as follows:

$$s(x, y) = \frac{\sigma_{xy} + c_3}{\sigma_x \sigma_y + c_3}, \tag{2.23}$$

where σ_{xy} is the covariance of pixel values between images x and y. For computing σ_{xy}, two pixels are considered to be a pair if they are at the same location, in different images. c_3 is a small constant to stabilize the division when the denominator is close to zero. Again, it can be shown that the maximum value is 1, and is reached only if $x = y$, meaning that the two images contain pixels dispersed identically around their respective mean at every location.

Finally, SSIM is obtained by combining Equations 2.21, 2.22, and 2.23 as follows:

$$\begin{aligned} \mathrm{SSIM}(x, y) &= l(x, y) \cdot c(x, y) \cdot s(x, y) \\ &= \frac{(2\mu_x \mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}, \end{aligned} \tag{2.24}$$

where SSIM(x,y) is a value between -1 and 1, with 1 being the highest score and meaning that the two images are identical.

In summary, SSIM is a great metric for comparing two images similarly to how human perception does it.

2.8.4 Fréchet inception distance

The Fréchet inception distance (FID), developed by Heusel et al. [11], is a perceptual score employed to assess the quality of generated content, specifically designed to evaluate GANs. More precisely, the FID compares the distribution of training examples used to train the network, p_{data}, against the distribution of generated samples, p_z, in the following manner:

$$d^2((m, C), (m_w, C_w)) = L_2(m, m_w) + \mathrm{Tr}(C + C_w - 2(CC_w)^{1/2}), \tag{2.25}$$

where m_w and C_w are the mean and covariance of the generated distribution, while m and C are the mean and covariance of the original training examples distribution. Tr refers to the mathematical trace.

Both distributions are usually approximated with the pre-trained Inception v3 network [26], selecting the last pooling layer's activations as the observation features. Because this layer returns 2,048 features, each image is represented by a 2,048-dimensional vector, and consequently, m_w and m, the feature means, are 2048×1 vectors, while C_w and C, the feature covariances, are 2048×2048 matrices.

The 2,048 activation features returned by the Inception network are assumed to follow a multi-variate Gaussian distribution, and are used to empirically estimate the first two moments of the generated samples, m_w and C_w, and training examples, m and C. Therefore, Equation 2.25 evaluates how close the means and variances of the two distributions are to each other. Consequently, a high score means that p_{data} and p_z are dissimilar, based on the distance between their two first moments.

In summary, the FID compares the means and variances to quantify how resemblant two distributions are. One of its main advantages over other methods is its ability to efficiently detect disturbances, while also being robust to noise and image distortions. Furthermore, the Inception v3 model is known to produce features important to human perception, which aligns the FID with human judgment.

2.9 Deep image prior

In 2020, Ulyanov et al. [28] showed that for image generation, training on a large number of examples is not necessary to properly learn realistic image priors and low-level image semantics; in most cases, the structure of the generator network, alone, is sufficient to achieve this. The rationale behind their work is that a network can be trained to reproduce the valid parts of a corrupted image, and doing so, it will likely learn sufficient low-level semantics specific to that image to also be able to complete the corrupted regions. More concretely, starting from a random input, and using the corrupted version of the image as ground truth, this task is formulated as an energy minimization problem as follows:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, E(f_\theta(z); x_0), \tag{2.26}$$

where θ is the set of model parameters, x_0 is the corrupted version of the image, z is the a randomly-initialized $3D$ tensor of the same spatial dimensions as x_0, f represents the network, and E is the energy function to minimize, which is specific to each application.

This idea of leveraging the network structure to learn image priors can be applied to a wide range of image restoration tasks such as denoising, super-resolution, and even inpainting. When applied to inpainting, the energy function to minimize becomes

$$E(x; x_0) = L_2(x \odot (1 - m), x_0 \odot (1 - m)), \tag{2.27}$$

where $\odot$ is the Hadamard's product, $x = f_\theta(z)$, and m is a binary mask, where 0's and 1's respectively denote valid and masked pixels.

Minimizing Equation 2.27 pushes the model to learn how to exactly reproduce the pixels of the valid context. Therefore, the network, instead of being an algorithm capable of generalizing to many different images, is reduced to a function with millions of parameters representing a portion of a given image. When the masked region is relatively small, the expectation is that this function can be extended to the missing area, which is a reasonable hypothesis, given that the content of this missing region will likely be consistent with the rest of the image in terms of colors, structure and texture. This technique, however, presents a few limitations. First, generating an output is extremely slow, given that the model needs to be trained from scratch every time. This makes it a non-practical approach, in the sense that it is impossible to use it in a photo editing software where users are expecting an almost real-time experience. Second, the masked region needs to be relatively small in comparison to the rest of the image for the network to capture its content. Third, the model does not learn to be creative, and can only complete an image with the patterns contained in it, which limits the usage of this approach.

Chapter 3

Test-Time Training for Inpainting

In this chapter, we will describe our approach. Given a convolutional neural network trained on a large dataset for the task of inpainting, we propose to finetune it on the test image, which is the user-provided image that contains holes to be filled.

3.1 Motivation

Our motivation is as follows. The goal of inpainting is to fill in missing pixels with plausible content. To this end, one can usually train a CNN on a large dataset of images. Then, given an unseen image with holes, the network will ideally have learned how to complete these missing regions from prior examples it has seen during training. Thereby, it is advantageous to use a training dataset that contains examples similar in content to the image a user provides. Indeed, when one needs to inpaint an image featuring faces, the CelebA faces dataset [20] is typically used for training. Similarly, for completing indoor or outdoor scenes, the Places2 dataset [38] is a very popular choice for training.

Given an image to fill, a possible approach is to find a similar one, for example by Web search as in the work of Hays et al. [9], and transfer appropriate content from that similar image to the user image. This technique, however, requires searching over a huge image collection, and furthermore, finding suitable portions to transfer to the new image is difficult. We observe that the user input itself typically already has a lot of content that could seamlessly fit inside the holes. Indeed, to a certain degree, the user input to complete is often its own most similar image, provided that the missing regions are not extremely large.

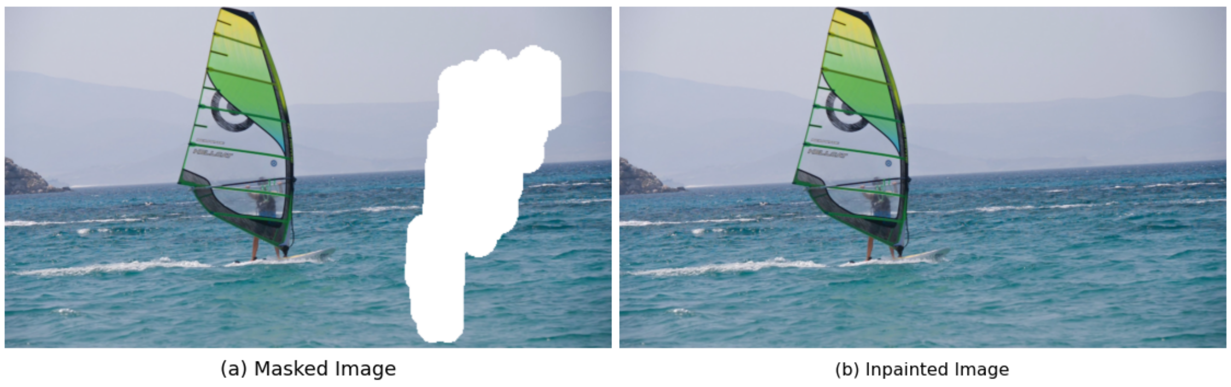


Figure 3.1: Example of a corrupted image where missing content can be inferred from valid context.

Let's consider the example above in Figure 3.1. This hole could probably be filled in a reasonable way by finding a picture on the Web of waves on a grayish sky background. However, finding such an image could be computationally expensive, and there is no certainty that colors will match perfectly. Therefore, the network will have to learn where to find the missing information, and how to adapt it. On the other hand, looking at the valid context of the input image, it is easy to see that the missing content can be completely borrowed from the valid context. In addition to removing the difficulty of finding a similar example, it also reduces the complexity of the task, since colors are already consistent with the rest of the image. As mentioned above, this is optimal when the masked area is not too large, and also when the image structure contains a few repetitive patterns.

Thus, we propose a test-time training approach for image inpainting. Given a CNN trained for this task on a large dataset, and a user image, we construct a small set of examples from the user image, and finetune the given CNN with it for a few iterations. We do not finetune for too long, because we want to avoid excessively suppressing what the network has learned initially when trained on a large collection of images. Yet, by training on a small number of images directly related to the user input, the hope is that the network will learn to fill holes specific to the user image better.

3.2 Test-time training

Also referred to as online learning, test-time training is primarily used for domain adaptation. Let's consider the fixed set of examples on which a CNN is trained. This set of examples forms an image distribution, and just like all other distributions, it has a domain. Domains in the image space, however, are extremely various and particularly hard to bound. Thereby, it is very likely that images at inference time might not pertain to exactly the same domain as training images, which can greatly degrade performance. A possible solution is to have a dataset for each very specific domain; however, data collection requires considerable efforts, and the same CNN would need to be trained on each single dataset, which would take an enormous amount of time. Instead, the user input at inference time often carries valuable implicit information about its domain. Normally, this is thrown away, hoping that the knowledge gained during training, alone, will be sufficient for the task at hand. Test-time training, on the other hand, exploits that information to adapt the CNN and the underlying domain of the data distribution on which it has been trained.

Taking advantage of implicit clues found in the input at inference time can take different forms. For example, let's consider the problem of monocular video depth estimation, which is the task of determining the depth of each pixel, for each frame of a video, based on 2-dimensional images. Depth is usually straightforward to infer when there are two or more views with different angles of the same objects, and when the camera's location in space is known. When all these conditions are met, depth can be retrieved in a rather simple way using geometric constraints. The difficulty in monocular depth estimation therefore lies in the absence of a second view. Because of the complexity of the task, CNNs are commonly used to address it. However, they only leverage the information presented at training time, discarding all additional clues the test-time input could contain. When considering videos, it is reasonable to assume that two frames close in time will likely display the same objects, while having different angles, but still with a similar camera pose. Leveraging this, and with the help of test-time training, one can formulate a loss from geometric constraints to quantify the exactness of the network's output at inference time, and use this to calibrate the parameters for the video at hand [21]. Other than monocular video depth estimation, online learning can be applied to a wide variety of problems, such as unknown object tracking in videos [16, 23], face identification [4], face detection [13, 36], super-resolution [24], and inpainting. More generally, applications for which additional and useful information can be extracted from the test-time input in an automatic manner are particularly well-suited for online learning.

Traditionally, in inpainting, a region of the image gets masked, and the network tries to fill it, guided by the knowledge it has gained from its training on a large number of examples. However, even at test time, the valid context of the input image can be rich in information, and relying solely on previous examples, the network might not be able to take full advantage of it. Indeed, finding correspondences between different parts of an image is important for completing missing regions, and these relations can vary a lot from one sample to the other. Inspired from work by others [21, 16, 23, 13, 27, 4, 36, 2, 3, 24], we encourage the network to exploit the valid context of the test-time image by adding an intermediate step between training and inference, during which the model further refines its learning by looking at the known areas of the input. Therefore, our contribution can be seen as a finetuning framework and an adaptation of online learning for inpainting.

3.2.1 Pre-trained model

Learning with a single example is not sufficient to comprehend the complexity of structures and textures in images. Consequently, to apply test-time training, the first step is to train an inpainting network as usual on a large set of examples. The exact details of how the model is pre-trained for our experiments will be given later in Section 3.3. This backbone network is crucial and should be able to generate plausible content where necessary, since the valid regions of the test input might not be sufficient alone to complete the corrupted areas in the case of complex structures.

3.2.2 Finetuning masks

A traditional inpainting network receives as input an image that contains two distinct areas, that is, the valid and user masked areas. In the case of our framework, we also need to define a third region. Indeed, the user masked region does not have a ground truth, and thus, cannot be used to compute the loss and optimize the parameters. Instead, a subset of the valid area is hidden with the help of an additional mask, and used for training at test time. Throughout this thesis, we will refer to the mask provided by the user as user mask, and the masks that we use to generate a finetuning dataset for inpainting as finetuning masks.

Since pixels that are hidden by the user mask do not convey any useful information, they should ideally be excluded from the input at test-time training. With square masks, this can be easily achieved by taking square regions around the user hidden area, and upscaling them to the original dimensions. Depending on the size of the user mask and the dimensions of the square regions sampled, this will return many "new" images that can be used for test-time training. However, most practical cases involve free-form masks, for which this strategy does not work.

In the case of free-form masks, isolating an undamaged region that can be upscaled to the original size is not trivial. Indeed, the mask can have any shape and span the image in any way. Because of this, finding regions with enough consecutive valid pixels is not trivial.

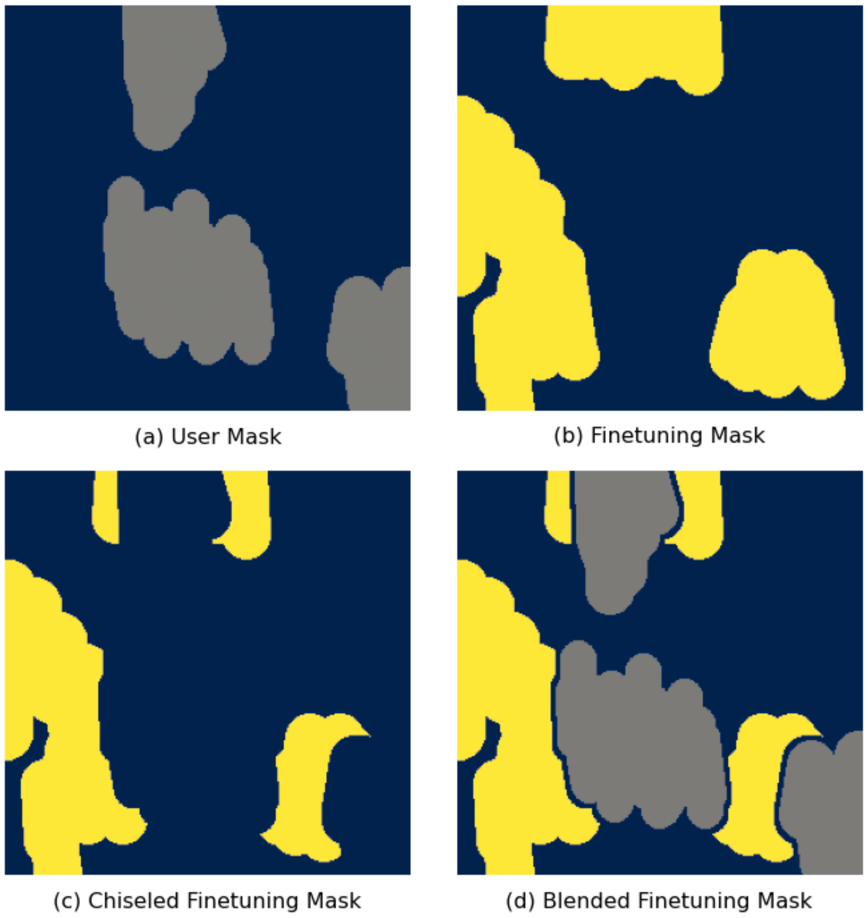


Figure 3.2: (a) Mask provided by the user. (b) Randomly sampled mask. (c) Randomly sampled mask with its areas overlapping test-time mask removed. (d) Result of (c) superimposed to test-time mask. The thin border of valid pixels between the yellow and gray areas is due to a binary dilation applied on the user mask.

For our experiments, we chose to randomly sample new masks as described in Section 2.3.2, until they meet certain criteria. More precisely, let's define U, the area covered by the user mask provided at test time, and F', the area covered by the finetuning mask randomly generated. Then, F' is chiseled in a way to remove all overlaps with U, which produces F. Formally, $F = F' \setminus U$. Figure 3.2(c) shows an example. Furthermore, we apply a binary dilation on U, resulting in U', to slightly expand the area covered by the user mask on all sides. Consequently,

$$F = F' \setminus U'. \tag{3.1}$$

This adds a thin border of valid pixels between F and U, which is important because immediate valid pixels are usually the most informative to fill masked regions. The result of this operation is shown in Figure 3.2(d). Finally, in our experiments, a sampled mask is retained only if the area percentage of the image it covers, after removing overlaps with U', is between α and β, two hyper-parameters.

Using the strategy described above, we are able to generate a finetuning mask that hides a subset of the valid context, without overlapping with the user mask. Nonetheless, pixels hidden by the user mask are still part of the input, which is not desirable because they do not have a ground truth. Consequently, they cannot be used to guide parameters optimization, and more importantly, they corrupt the image, affecting the model's ability to learn a proper representation and extract true features. To handle this, we fill user masked areas with placeholder values, and guide the model not to consider these regions through the loss function. More details will be given later in Sections 3.2.3 and 4.4.

3.2.3 Loss function and optimization

Another important consideration is the loss function. Normally, in inpainting, one constructs losses for the foreground and background separately, and then combines them into a final result using a weighted addition. For our framework, we use F as the foreground (see Equation 3.1), and define the background as image pixels that are outside the user and finetuning masks, that is

$$B = (F \cup U)^c. \tag{3.2}$$

In Figure 3.2(d), F and B correspond to the blue and yellow regions, respectively. Therefore, the region delimited by the user mask, U, is completely disjoint from both F and B. This is intended, since we do not have a ground truth for pixels inside the user mask, and thus we cannot optimize the model parameters based on these output values. Consequently, even when applying the loss function on the foreground and the background, the network does not learn to fill U.

For our experiments, we use the L1, structural, perceptual, and adversarial loss functions as defined in see Section 2.7, and apply each of them separately on the foreground (F) and background (B) regions. This is straightforward to achieve for the first two functions, since they consider each pixel individually; the last two, however, treat the image as a whole.

To apply the perceptual loss on F only, we replace pixels in F^c with their original input values before computing the loss. We call the result of replacing pixels in F^c with their original input value the completed output. Assuming the finetuning mask is binary, where the values 0 and 1 respectively designate pixels in F^c and F, this can be expressed as

$$\text{completed output} = \text{finetuning mask} \times \text{output} + (1 - \text{finetuning mask}) \times \text{input}. \tag{3.3}$$

Consequently, the loss considers F exclusively, since only values within the region F of the completed output are a function of the model's parameters; all others are constants whose derivative is equal to 0, thus not impacting training. Following this logic, applying the perceptual loss on B should be done by replacing pixels in $F \cup U$ with their original input values. Again, assuming the user mask is binary, where the values 0 and 1 respectively designate pixels in U^c and U, this can be expressed as $(1 - (\text{finetuning mask} + \text{user mask})) \times \text{output} + (\text{finetuning mask} + \text{user mask}) \times \text{input}$. However, we found out that looking exclusively at the background produced unstable results for this loss. Instead, we only replace pixels in U when applying the perceptual loss on B, which stabilizes training. We refer to this as the raw output, and it can be defined as follows:

$$\text{raw output} = (1 \text{ - user mask}) \times \text{output} + \text{user mask} \times \text{input}. \tag{3.4}$$

Since we employ free-form masks in our experiments, we use a patchGAN discriminator as defined in Section 2.4.1 to compute the adversarial loss. Consequently, the inputs for this loss, the fake and real images, are divided into patches. For the real image, the background-foreground distinction does not apply. Moreover, it contains an area masked by the user. Therefore, we compute the adversarial loss on the real image by downscaling the user mask with average pooling to the dimensions of the discriminator's output, and only selecting patches where more than 50% of the pixels are in U^c. On the other hand, for the fake image, we can apply the background-foreground distinction. For the foreground, we apply the discriminator on the completed output (see Equation 3.3), and similarly to what is done for the real image, we downscale the finetuning mask with average pooling to the dimensions of the discriminator's output, and only select patches where more than 50% of the pixels are in F. For the background, we apply the discriminator on the raw output (see Equation 3.4), and select patches having more than 50% of their values in U^c.

Finally, network parameters that should be finetuned need to be chosen carefully to avoid forgetting prior training and overfitting to the known regions of the test image. We experiment with different sets of parameters by freezing earlier layers of the model. Indeed, these layers are given an input that is still close to the original input image, and thus, they focus on low-level features and granular details like colors, texture, and structure. These characteristics can usually be handled similarly for all images of the same distribution. On the other hand, later layers generally deal with a significantly modified input, called the latent representation, which is better for analyzing and extracting high-level features. These features are more specific to each image, hence why it is advantageous to finetune parameters contained in later layers rather than those contained in earlier layers.

3.3 Model architecture and pretraining

To conduct our experiments, we use a simple, yet effective, three-scale attention network paired with a patchGAN discriminator (see Section 2.4.1), inspired from [33]. The encoder consists of 3 convolutions to downsize the input by 4, followed by 8 residual blocks, while the decoder is built with 3 contextual attention layers, each followed by a convolution-deconvolution duo to restore the original image dimensions. The discriminator comprises 5 convolutional layers, and returns a $32 \times 32 \times 1$ output map when given an RGB image of size $256 \times 256 \times 3$ as input. The total number of parameters is 17,849,825. The input of the generator is the concatenation, along the channel axis, of the image, edge map, mask, and image gradients. Pixel values in the image are scaled to the range [-1, 1] for standardization. The edge map is obtained with Canny edge detection, while image gradients are computed with sobel filters. The network outputs the reconstructed RGB image. See Figure 2.7 for all the details.

The model is pretrained on 100,000 256×256 images of Places2 [38] (1000 images per category, for the first 100 categories) for 20 epochs on a single RTX 2080 Ti GPU, for a total runtime of 11 hours. Masks are randomly generated and of free-form shape. L1, structural, perceptual, and adversarial losses are all applied on the background and foreground of the final output with a respective weight of 1, 1, 0.1, and 0.4. The hyper-parameters of the structural loss, α and β, are respectively 0.1 and 20. A constant learning rate of 1×10^{-6} is used throughout all 20 epochs. The choice of hyper-parameters is based on the implementation in [33].

We decided to use this model because it has a relatively small number of parameters and simple architecture, while being easy to train and returning close to state-of-the-art results. Moreover, during test-time training, the main objective is to establish relationships between different parts of the image that can be reused to complete the user input. With three attention layers, the model selected is the ideal candidate for this type of learning.

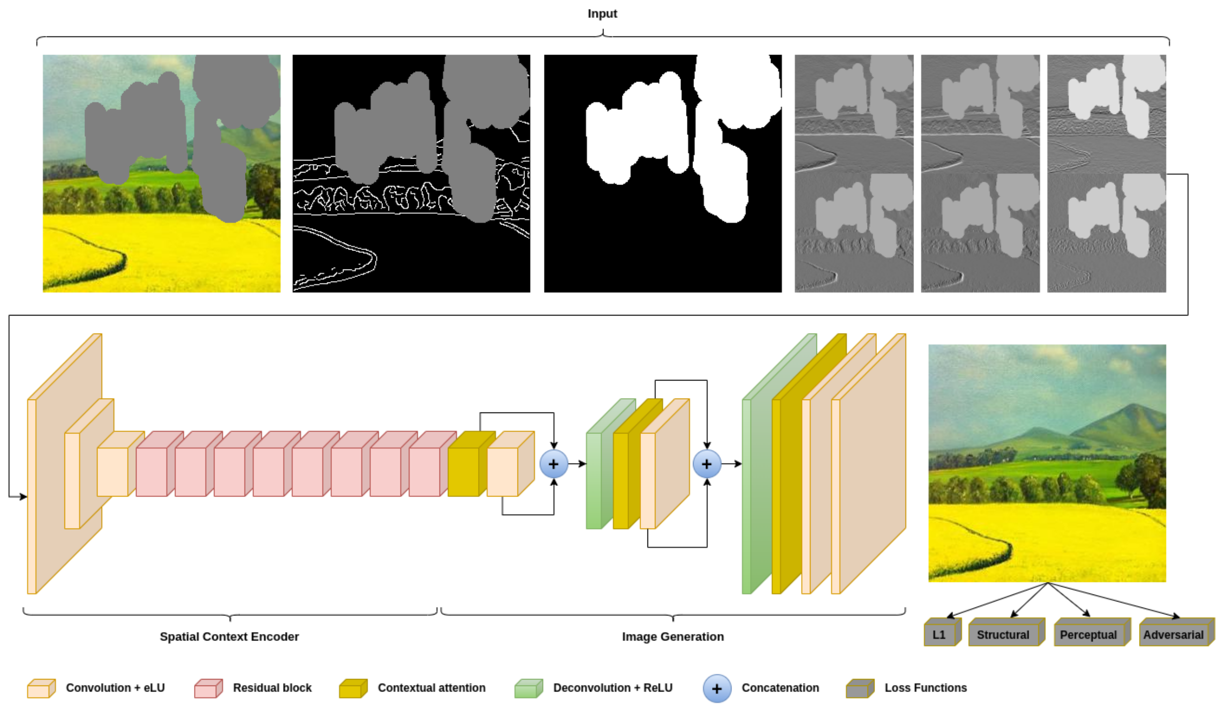


Figure 3.3: Overview of our model architecture, inspired from [33]. The input is formed by the concatenation of the masked image, masked edge map, mask, and masked gradient fields. There are 6 image gradients in total, 1 per axis (vertical and horizontal) per channel. They are downscaled in the above figure because of space constraints, but they are in reality of the same size as the image. The output is the completed image, on which loss functions are applied.

Chapter 4

Experimental Results

In this chapter, we present the results of our finetuning framework for different experiment setups. At the end, we also show a few visual outputs to demonstrate the improvements that can be achieved with this technique.

To evaluate the results of our framework, we use the L1, PSNR, SSIM and FID evaluation metrics, described in Section 2.8, that we compute every 5 steps for efficiency. For consistency, we randomly simulate user masks at the beginning, one for each image, and reuse them for every experiment. On average, these masks hide 28.86% of the image. For the sections that follow, we define a finetuning step as the processing of one batch of 4 images. By "processing", we refer to the task of computing the loss function on the network's output, and then updating parameters through gradient descent. Since our framework finetunes on the input given at test time, the image is always the same for all steps; however, they each have their own finetuning mask, and can have different augmentations (see Section 4.3).

4.1 Baseline

For our experiments, we use the Places2 dataset [38], which is a large collection of 10 million scene photographs, representing different types of environments that can be encountered in the real world. Each photograph is labelled with one of 365 scene semantic categories, which can in turn be classified as either indoor, outdoor natural, or outdoor man-made. Unless stated otherwise, we run the experiments of the next sections on 116 randomly selected images from Places2 to allow efficient testing of many different settings. Depending on the experiment, test-time training takes around 100 seconds per image on a single RTX 2080 Ti GPU.

As mentioned earlier in Section 3.3, the backbone model utilized is a network pre-trained for 20 epochs on $256 \times 256 \times 3$ images, whose input is the concatenation of the image, edge map, mask, and image gradients. With it, we build our baseline in the following way. The user masked areas are filled with random pixels. The learning rate is initially set to 1×10^{-6}, and divided by 2 every 50 steps. Proceeding as described in Section 3.2.2, different free-form finetuning masks are generated for all images, even those within the same batch. We retain a sampled finetuning mask only if the area that it hides, after removing overlaps with the user mask, is between 15% (α) and 25% (β) of the total surface of the image. Following our description in Section 3.2.3, we apply the L1, structural, perceptual, and adversarial losses only on the foreground. Finally, test-time training is performed for 150 steps.

For all the experiments that will be presented below, we start from the baseline and make small adjustments to test different assumptions and finetune specific parameters. At the end, we present a few combinations that regroup modifications from our experiments that returned the best improvements.

4.2 Finetuning mask generation strategies

Throughout prior experimentation, it has been found that inpainting networks, in general, are very sensitive to the shapes and sizes of masks. In this section, we test different finetuning mask generation strategies, and present the results below in Figure 4.2.

4.2.1 Augmented finetuning masks

The objective of test-time training is to adapt the model to the image it will later have to complete. Here, we further orient its learning by also giving it explicit clues about the shape of the user masked area. To do so, we enforce shape consistency between user and finetuning masks by taking the former, which is always the same for all steps, and augmenting it with random flips, translations and rotations to produce finetuning masks. For an illustration of these augmentations, see Figure 4.6. Even though the resulting finetuning mask is positioned differently than the user mask, and its shape might slightly differ because of the chiseling operation, it remains a good approximation of the user masked area's shape. This not only allows the network to learn how it should use the available valid context, but it also allows it to learn how to deal with this specific user mask shape.

The results presented in Figure 4.2 show that creating finetuning masks by augmenting the user mask, instead of randomly generating them, does not produce a significant improvement. A possible explanation for this is that random rotations and the chiseling operation excessively alter the user mask, which does not let the model recognize the shape and learn from it.

4.2.2 Full-image finetuning masks

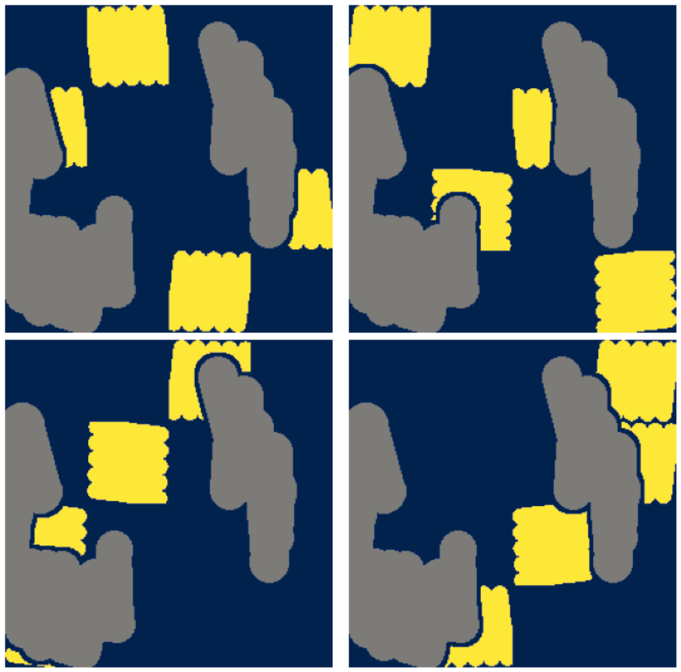


Figure 4.1: Full-image finetuning masks for a single batch of 4 images. Together, yellow regions cover the entire valid context area of the image. The thin border of valid pixels between the yellow and gray areas is due to a binary dilation applied on the user mask. The generated finetuning masks (yellow) are not perfectly square shaped. This is because the backbone model was pre-trained with free-form masks, so we tried mimicking a similar shape to avoid destabilizing the network by introducing new square shapes.

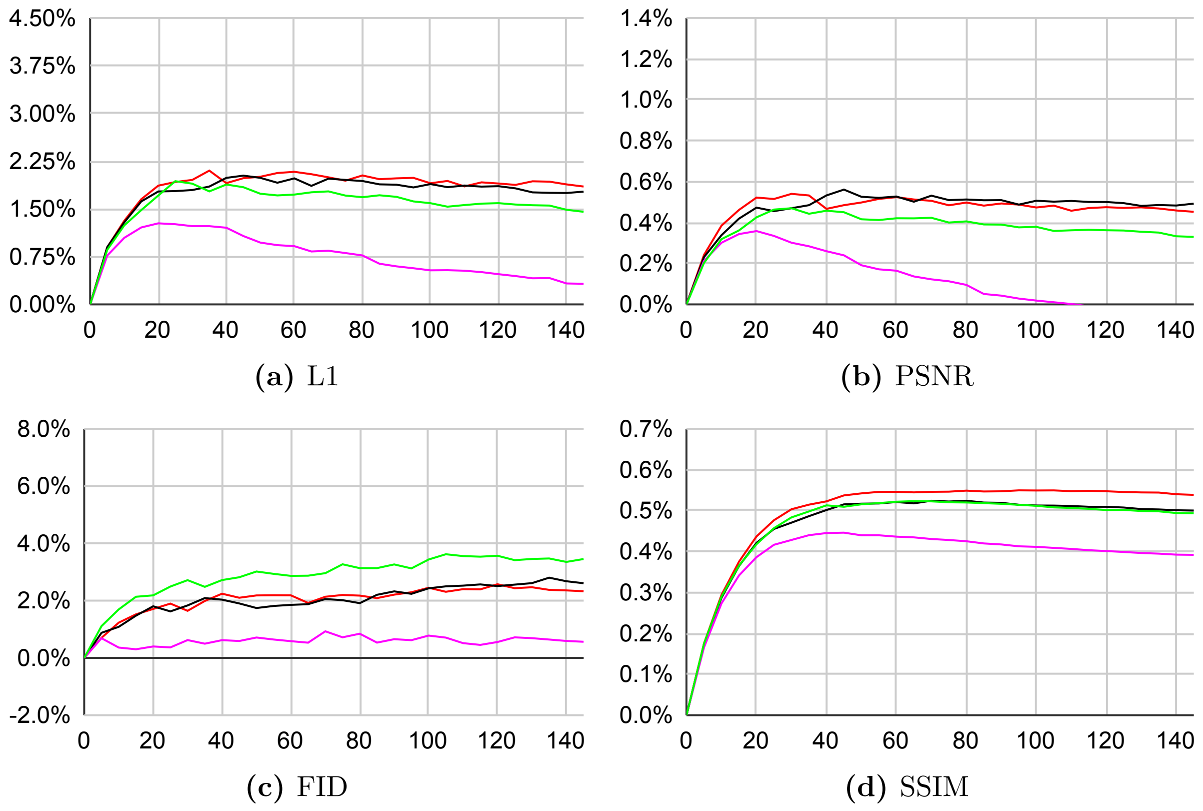
Ideally, our framework should gain a uniform understanding of the valid context during the finetuning phase. However, because finetuning masks are randomly generated, the same regions might or might not get hidden for many consecutive steps, which unbalances the whole learning process by making the model overfit to one specific region, and then shifting its attention to a different part of the image. To stabilize test-time training, and also to make sure the network focuses on all available parts of the image, we use full-image finetuning masks. In essence, we separate the image into 16 square parts of equal area, and randomly hide 4 of them in each image of the same batch, such that the union of masks in one batch covers the entire image. This way, the model is obligated to consider all pixel values at every step, thus preventing overfit to one region and balancing the focus of the network uniformly across all areas.

In Figure 4.2, we can see that this experiment produced poor results, below the baseline. As stated previously, inpainting networks are very sensitive to the shapes and sizes of corrupted areas. Using disjoint almost-square shaped finetuning masks that span the entire image might be too different from the free-form masks that the model was accustomed to see when pre-trained; therefore, its prior knowledge might not have been as useful as it could have been.

4.2.3 Larger finetuning masks

The foreground loss, as defined in Section 3.2.3, only considers the values inside the finetuning mask. For our baseline, we exclusively use this loss, which means that all the parameters of the network are optimized based solely on the subset of values indicated by the finetuning mask. An obvious downside of smaller finetuning masks is therefore that only a small portion of the image dictates how all parameters should be optimized at each step, which causes the gradient to vary a lot from iteration to iteration. In other words, if the network is only given a small portion of the image, then it is easy to learn specifically how to complete this area given that it doesn't have too many values to predict, but this might not be appropriate for the next portion. Indeed, small patches can be very different from each other, even within the same image, and, therefore, the network constantly gets adapted, and never converges to a global understanding of the image.

Instead, larger masks can help stabilize the gradient, and allow the model to better understand the global context of the image at each iteration. Because we are using relatively small finetuning masks for our baseline (covering between 15% and 25% of the surface of the image), this experiment tests larger finetuning masks by setting the hyper-parameters α and β, defined in Section 3.2.2, to 23% and 30%, respectively. As a result of this change, L1 and PSNR slightly decreased below baseline, while SSIM stayed roughly the same, and FID improved (see Figure 4.2). The latter is expected, as perceptual scores like FID are greatly affected by the coherence of the image at a global level, and small finetuning masks do not allow the model to focus on large regions. A possible explanation for the drop in L1 and PSNR could be that since the network looks at more pixels at each iteration, it might not be able to focus its learning on specific regions, and consequently, it fills the holes with average values, producing less precise results.



⬛ Baseline
🟥 Augmented finetuning masks
🟪 Full-image finetuning masks
🟩 Larger finetuning masks

Figure 4.2: Evaluation of different types of finetuning masks. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

4.3 Random augmentations

CNNs need to be trained on a large number of examples to learn adequately. Nonetheless, collecting and labelling data requires considerable manual effort, which is why it is a very common bottleneck in deep learning, and one of the main reasons why neural networks took so long to break through. Random augmentations are a direct solution to this problem. They consist in small modifications to an image, such that the result still looks realistic and could have been produced in practice under different viewing conditions, such as a varying camera angle, zoom, or luminosity. This is very useful because it allows to artificially create new examples that can be added to the dataset without any additional effort.

Random augmentations are well-suited for test-time training in inpainting, because they help compensating the lack of data, and here the network is limited to a single image, the user input. Indeed, we want the network to focus solely on the image it will have to complete. Therefore, complementing the user input with other images should be avoided, unless they are clearly related, since it would make the task of learning specific parameters adapted for the image at hand harder. Instead, we randomly augment the user input. This way, the network does not overfit to the valid context region because it is given a different version of the same input at each step, and it learns only with relevant information since the input is still similar to the original user image. In the next sections, we individually apply each random augmentation and present how they affect our framework. More precisely, we present the effects of random augmentations altering the appearance, and the effects of random augmentations changing the positions of pixels. In all of the following experiments, we apply the transformations to images whose pixel values have been scaled to the range [0, 1]. After augmenting them, we rescale them to the range [-1, 1] as is done for pretraining.

4.3.1 Random augmentations altering appearance

Random crop

Random crop consists in creating a "zooming" effect to a specific region of the input. More precisely, it is achieved by randomly selecting a smaller square portion of the original image, and upscaling it with an interpolation method to the original image dimensions. For this experiment, we apply random crop 60% of the time, and use a square window size of 128×128 along with nearest neighbor interpolation. We apply the same crop to the user and finetuning masks, and retain the augmentation only if the surface of the cropped area covered by the finetuning mask alone is greater than 10%, and the total surface of the cropped area covered by both the finetuning and user masks is less than 65%. If these conditions are not met, we resample a new crop region.

Random brightness

Random brightness is the operation of darkening or brightening the image. It works by scaling all pixel values by a factor close to 1, and then clipping them to the range [0, 1]. Therefore, if the factor is greater than 1, all values get closer to 1, which brightens the image. Similarly, if the factor is less than 1, all values get closer to 0, which darkens the image. For this experiment, we utilized random brightness 50% of the time, and randomly sampled a factor between 0.7 and 1.3.

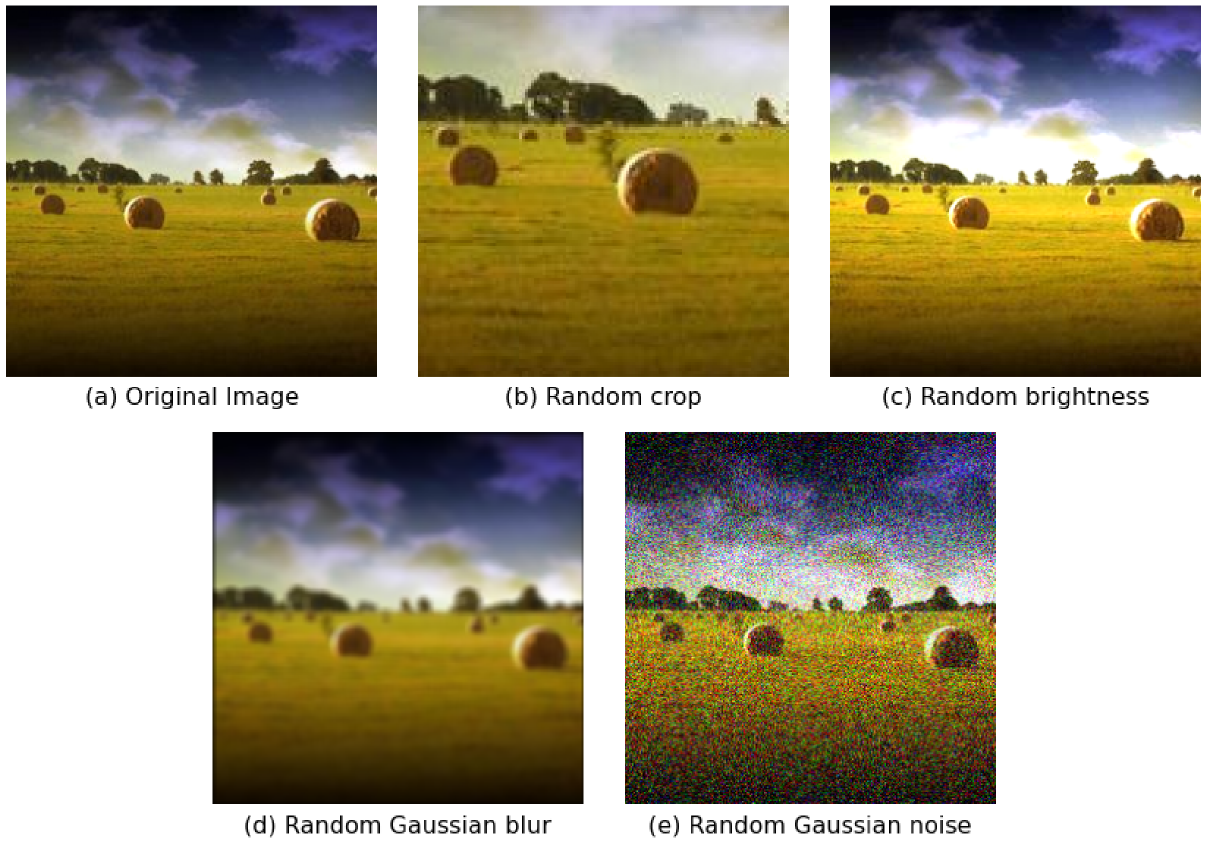


Figure 4.3: Example of random augmentations affecting the appearance of the image.

Random Gaussian blur

As its name suggests, random Gaussian blur blurs the image by averaging the pixel value at each location in the image with the values of neighbor pixels. This is implemented by convolving a 2-D Gaussian kernel with the original image. The size of the kernel determines how many pixels in the neighborhood of the location are considered, while σ, the standard deviation, controls the weight distribution, where increasing σ increases the weight given to neighbors and decreases the weight of the current pixel value. For this experiment, we apply random Gaussian blur 50% of the time, where the kernel size and σ are both randomly sampled in the intervals [5, 8], and [3, 6], respectively.

Random Gaussian noise

Random Gaussian noise adds random values sampled from a Gaussian distribution to each location in the image, which produces random and spontaneous variations of colors and brightness at the pixel level. The parameters μ and σ of the Gaussian distribution from which random values are sampled controls the intensity of these variations. For this experiment, we used a mean (μ) of 0, and a standard deviation (σ) of 0.2.

Results

Figure 4.4 shows the results of individually applying random crop, random brightness, random Gaussian blur, and random Gaussian noise to the input image during test-time training. L1, PSNR, SSIM improved by a small amount, except for random Gaussian noise. For random Gaussian blur, each resulting pixel contains, to a certain degree, information about its neighbors, which can help the network to efficiently make use of the valid context. Because of the upscaling, the output of random crop also contains blur, which, in a similar way to random Gaussian blur, can explain the small improvement. For random brightness, the proportions between pixel values are preserved, except for values going below 0 or above 1; therefore, it creates more examples, without requiring substantial adaptation from the network, since it only needs to learn one additional parameter, the scaling factor. Finally, the sudden shifts in pixel intensities that random Gaussian noise creates could explain why L1 and SSIM underperformed in comparison to the baseline.

FID consistently did worse than the baseline. Indeed, FID tries to quantify how plausible an image is to human eyes. However, these random augmentations all transform the image in a way that looks less natural, thus negatively impacting FID. This is true even for random crop, since the interpolation produces blur, which is similar to the effect of random Gaussian blur, but in a less pronounced way.

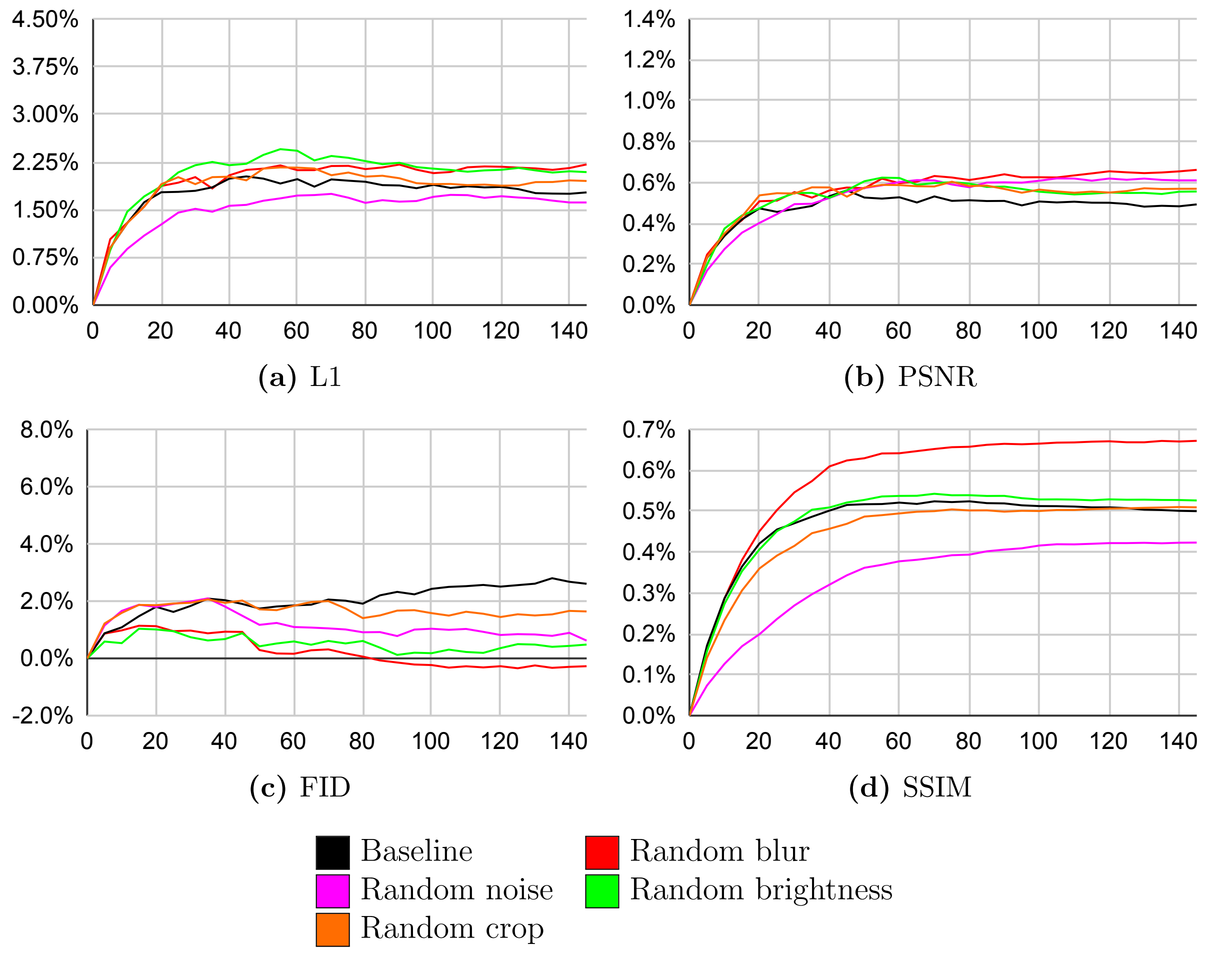


Figure 4.4: Evaluation of random augmentations affecting image appearance. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

4.3.2 Random augmentations changing the locations of pixels

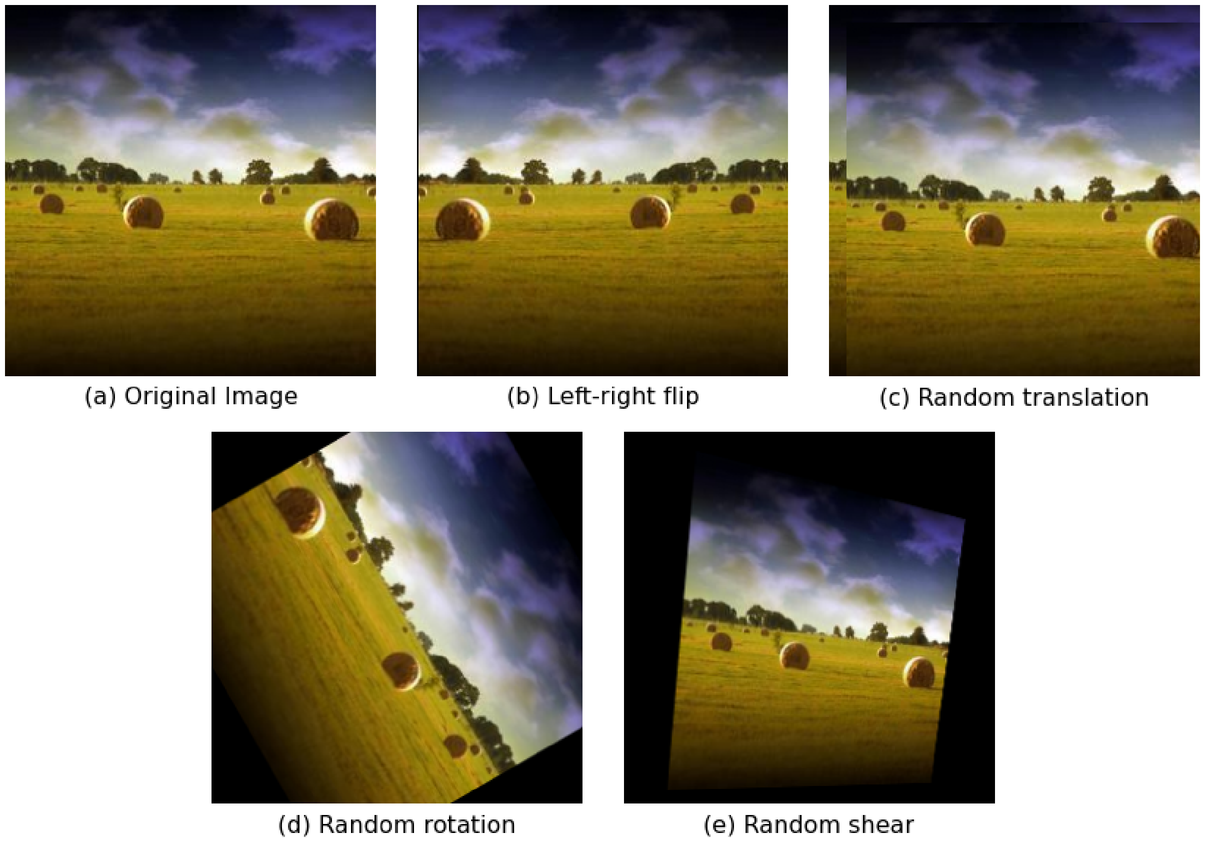


Figure 4.5: Example of random augmentations changing the position of pixels in the image.

Unlike the random augmentations presented above, random augmentations repositioning pixels in the image do not modify pixel values in any way; they rather change their locations in the image. These augmentations can be implemented with affine transformations. Affine transformations are a very common technique in computer vision, defined as geometric transformations that preserve parallelism and ratios of distances. In other words, even after applying an affine transformation, parallel lines will remain parallel, and points pertaining to the same line segment will stay at the same relative distances from each other. Affine transformations for 2-D images are characterized by a 3×3 matrix of the form

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}.$$

Consequently, there are 6 parameters that can be adjusted to produce different transformations. Once the parameters are set, the homogeneous 3-D location of each pixel in the image, $\begin{bmatrix} x & y & 1 \end{bmatrix}^T$, is left-multiplied by this matrix, which returns a new location. Then, pixel values are simply copied to their new locations, producing a general shifting effect, while preserving the visual content.

Left-right flip

A left-right flip produces a mirror effect by inverting pixels across the vertical axis in the center of the image. It can be done by applying an affine transformation whose transformation matrix is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For this experiment, we use a left-right flip 50% of the time.

Random translation

A random translation translates the image along the x and y axes. Its associated affine matrix is

$$\begin{bmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{bmatrix},$$

where v_x and v_y are respectively the distance of the shift in x and y. This transformation brings some of the translated pixels outside of the image bounds, thus leaving borders of empty values on two sides of the image. We fill those pixels with their original values, which produces the effect of two replicas of the same image stacked on top of each other, with one slightly shifted. For this experiment, random translation is applied 100% of the time, with the shifts in x and y both randomly sampled in the interval [-12, 12], including 0.

Random rotation

A random rotation rotates the image around its center origin (middle point) using the affine matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

where θ is the angle in radians. We apply two sets of random rotations. In the first experiment, we apply random rotation 100% of the time, and randomly select a rotation angle (in degrees) from the set {0, 90, 180, 270}, which means there is a 25% chance that the image will stay the same if the angle 0 is chosen. For the second experiment, we apply random rotation 50% of the time, and randomly select a rotation angle (in degrees) from the set {30, 60, 90, ..., 270, 300, 330}.

Random shear

A random shear works by adding a multiple of the location of y, sh_x, to the location of x, and a multiple of the location of x, sh_y, to the location of y. It is represented by the following affine matrix:

$$\begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For this experiment, we apply random shear 55% of the time.

One side effect of affine transformations is that they can map certain pixels to a location that is outside of the image frame, thus leaving "empty" pixels inside the frame. For random translation, we handle this by setting these empty pixels back to their original value. For random rotation and random shear, we tested two different approaches. First, we simply filled empty pixels with a value of 0, which is neutral, but at the same time confusing for the network since pixels inside the finetuning mask also have a value of 0. Thus the second approach was to fill empty pixels with random values, such that the network could easily distinguish them from the ones indicated by the finetuning mask. However, filling with 0's outperformed random values, so the results presented below are for the experiments where empty pixels are filled with 0's.

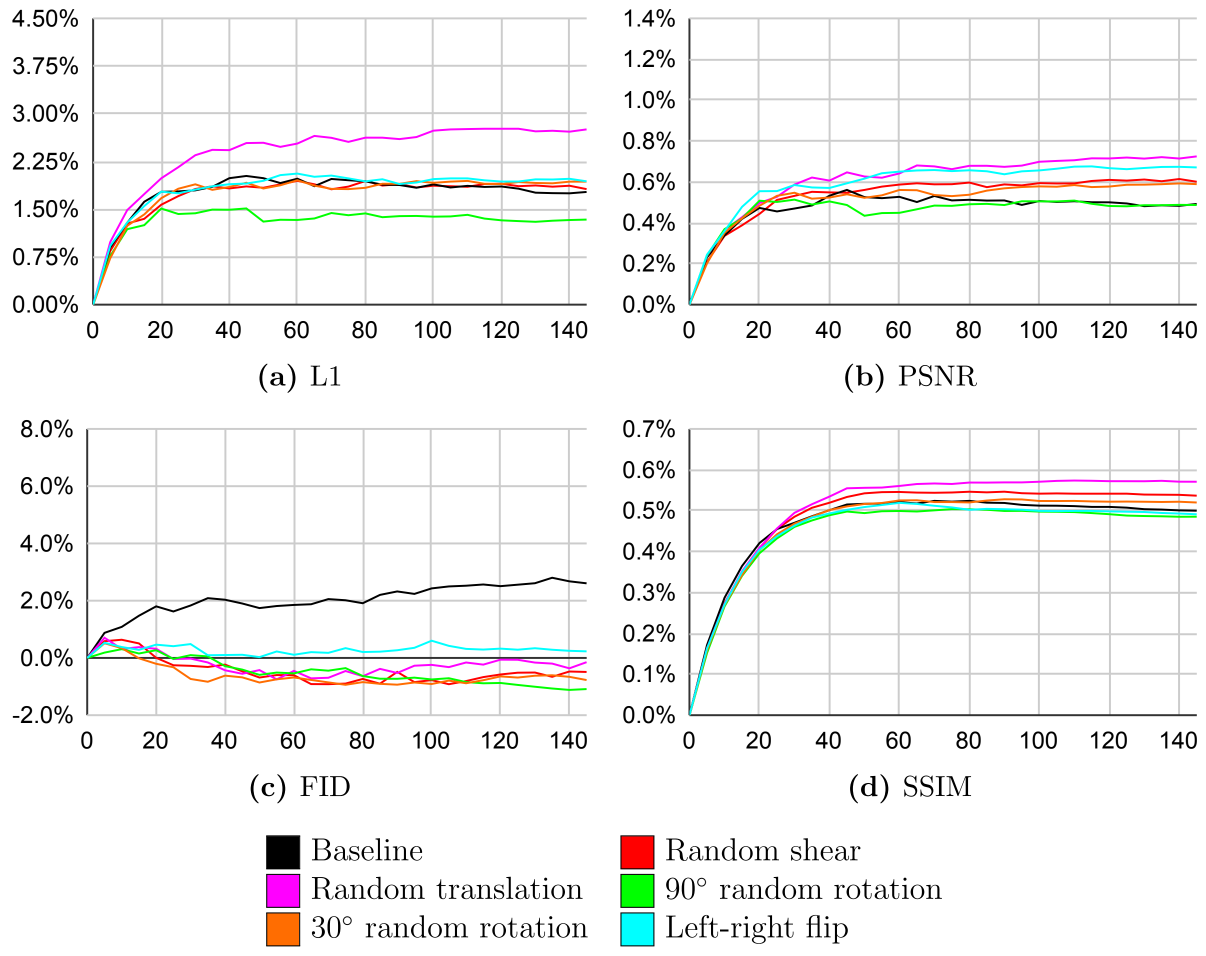


Figure 4.6: Evaluation of random augmentations changing the location of pixels in the image. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

Results

Figure 4.6 presents the results of applying each of these transformations individually to the input at test time. The most noticeable effect is the deterioration of the FID. As explained previously, FID quantifies the plausibility of an image, and small artificial changes like random augmentations can only diminish it, since they usually produce images that are less likely to have been produced in practice. Nonetheless, left-right flip produces plausible results, and yet, its FID score is much lower than the baseline. A possible explanation for this would be the increase of the sample variance that results from artificially creating new examples. Indeed, sample variance is defined as $S^2 = \frac{\sum(x_i - \bar{x})^2}{n-1}$, which means that training on additional images that are different will increase the numerator, but because the evaluation is still performed on the same number of examples, n stays the same. When looking at Equation 2.25, we can see that the FID score gets worse as C, the covariance matrix of features in the set of images on which it is computed, increases, and since C is directly related to the sample variance, it can explain why the FID score decreases.

For other metrics, the 90 degrees random rotation experiment is the one that performs the least well, even worse than 30 degrees random rotations. This was somehow expected as we are using natural images, for which ordering of colors is similar from one sample to the other. For example, there is often blue at the top to denote the sky, while green and brown, for the grass and ground, are commonly at the bottom. Random rotations reverse this ordering, which can hinder the network's ability to understand the global context of the image based on its prior training. On the other hand, random translation shows the best improvements, which could mean that it properly balances fidelity and novelty: the transformed image remains similar enough to the original one for the model's pretraining to be relevant, while being different enough to be considered as another example. Finally, for other experiments, results are very close to the baseline.

4.4 Filling user masked areas

It is important that our framework does not rely on the user masked area when learning to complete finetuning masked regions. Indeed, there is no ground truth for the user masked area, and it does not contain any additional information, which means that considering this region at test time can only worsen the result. This can be partially achieved by defining a loss function that explicitly excludes pixels outside the finetuning mask as described in Section 3.2.3. However, this is not sufficient as the model contains many attention layers that are aimed at considering distant information, and also because of the downsampling happening in the encoder, which blends pixel information and makes it harder for the network to distinguish user masked values from the valid values. Therefore, it is necessary to teach the model to recognize user masked pixels and to not use them, which can be done by adding explicit clues to the user input. For our baseline, we fill the user masked area with new pixels that are randomly generated for each sample, which is fast and easy. Consequently, the network learns that it should disregard these values, as using them in any way does not produce consistent and plausible results. Nonetheless, a downfall of adding random pixels to an image, even just in a bounded region, is that it creates noise at the global level and is detrimental for the perceptual scores. Below, we explore different ways to fill user masked areas, with the intention to provide explicit clues to the network without affecting evaluation scores.

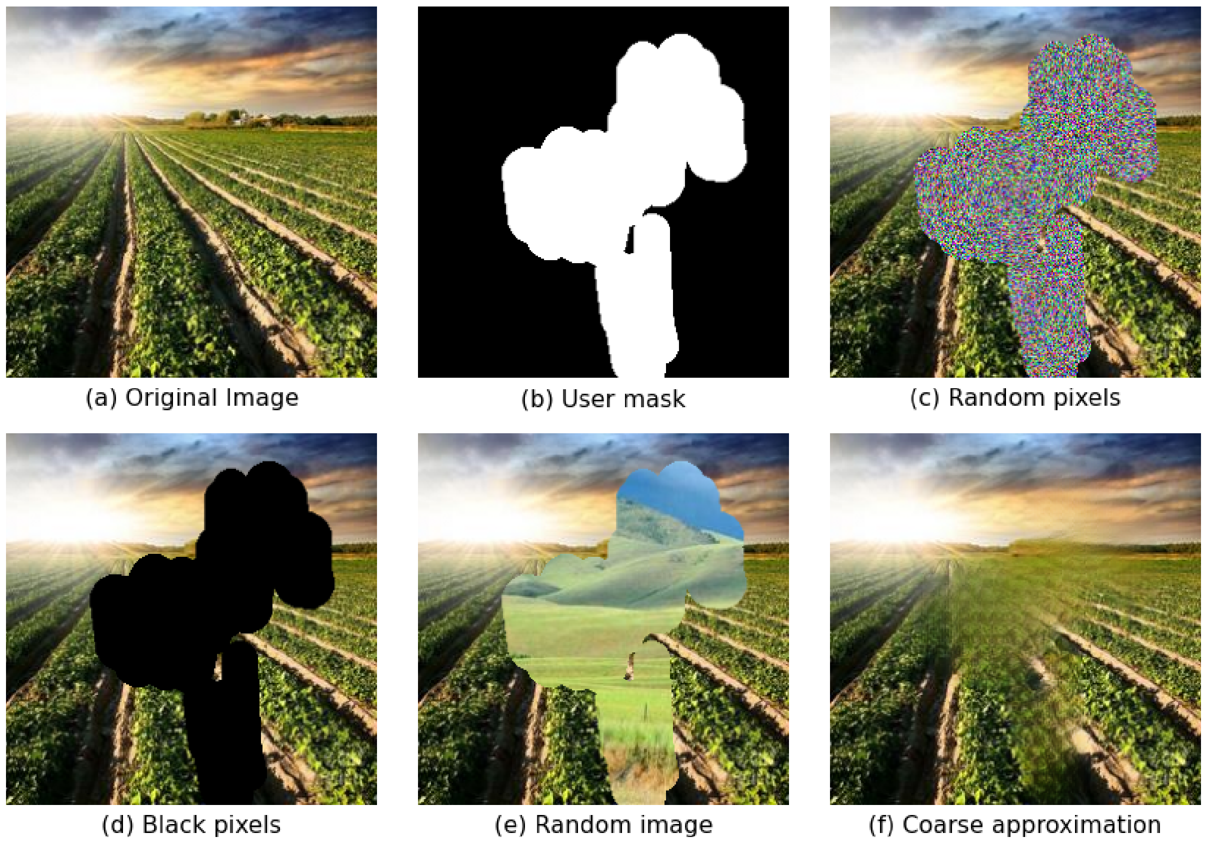


Figure 4.7: Different techniques to fill the user masked area.

Black pixels

The first alternative we test consists in filling the user masked area with black pixels, which is a naive and simple procedure. Unlike random values, it is neutral and does not add too much noise to the image. However, because these values stay the same for each sample, filling with black pixels does not provide good indications to the model. Instead, because these values do not vary and are not marked by the given finetuning mask, the network is likely to mistake the user masked region for a real one, and think that such a sudden cluster of black pixels is normal and expected in this image. Moreover, black pixels are also used to fill the finetuning masked area during test-time training, which could confuse the network even further.

Random image

As a second alternative, we fill the user masked area with a random image from the same data distribution. This technique possesses the advantages of filling with black pixels and random pixels, while still being efficient to apply. On one hand, it does not add excessive noise since the content is real, and on the other hand, it is different for each sample, which tells the network that it cannot trust this information to be reliable as it often changes and is likely not compatible with the rest of the image. Nonetheless, this incompatibility might alter the ability of the network to understand the global context of the image, as there are two completely different scenes in the same frame.

Coarse approximation

Lastly, we also fill the user masked area with a coarse approximation, which is obtained by applying the pretrained network on the user input, and carving out the region of the output that corresponds to the user masked area. Assuming the user mask is binary, where the values 0 and 1 respectively designate valid and hidden pixels, this operation can be defined as $\text{mask} \times \text{output} + (1 - \text{mask}) \times \text{input}$. The coarse approximation is computed with the current state of the network. Therefore, at step 0, the raw pretrained network is used, and at step 20, the pretrained network finetuned for 20 steps is used. This way, the network can learn incrementally by using the knowledge it has gained during the last steps to gradually produce a more accurate approximation of the user masked area. Nonetheless, computing a coarse result by running the model in inference mode is computationally expensive. Moreover, it creates a loop of errors, in which the model is more likely to reproduce the same mistakes as opposed to learning from them. Indeed, there is a high chance that the coarse approximation will contain undesirable artefacts, and since the model is not explicitly told that this is incorrect, it might treat it as valid context, which adds bias to the output.

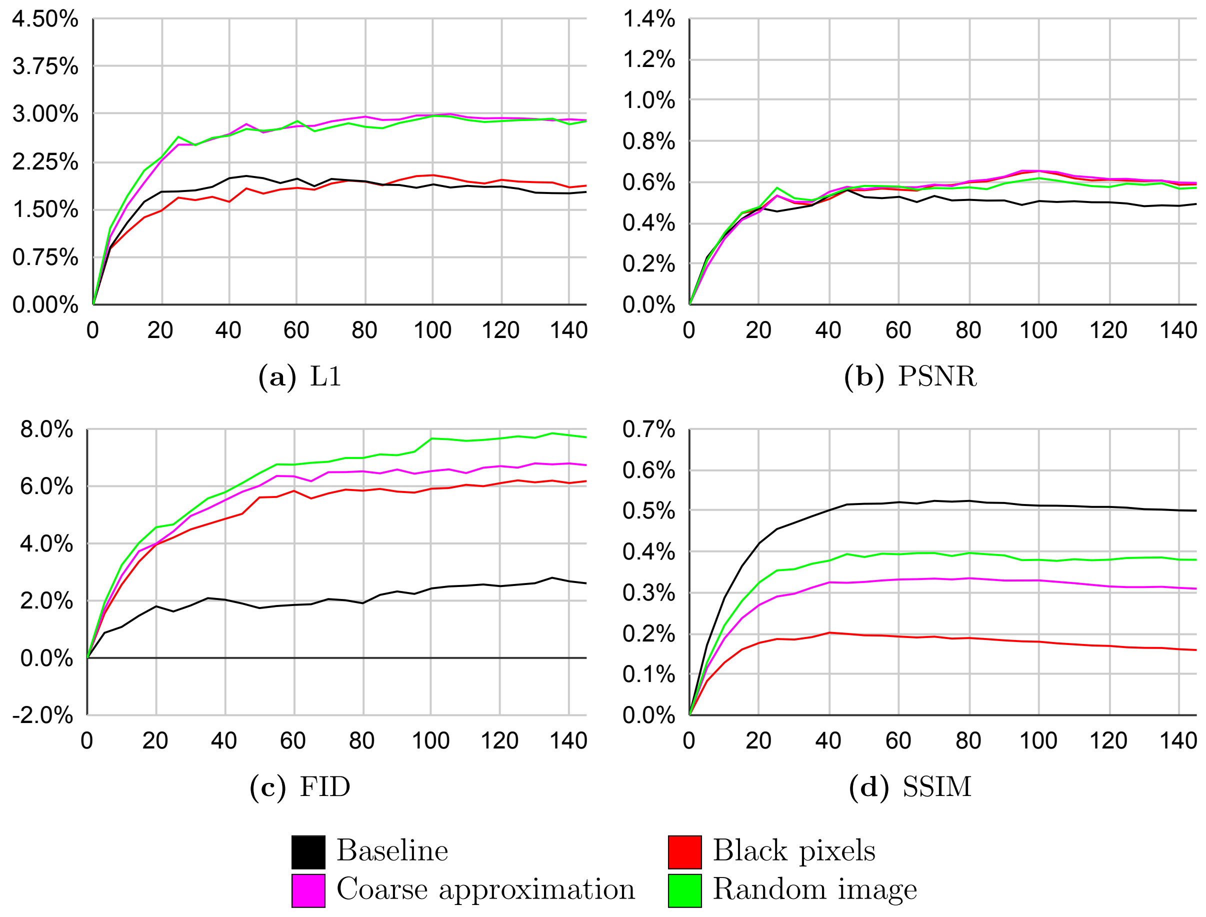


Figure 4.8: Evaluation of different types of filling for the user masked area. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

Results

The results of applying these different types of filling for the user masked area are shown in Figure 4.2. FID shows the biggest improvement when compared to the baseline. This is expected as the baseline completes the user input with random values, which, even if only applied to a targeted region in the image, adds noise at the global level causing perceptual scores like FID to decrease. Among the three types of filling tested above, random image performs the best for FID, which, again, is not a surprise, given that completing the input with a random image divides it into two parts that are real, and while they might not fit together in terms of colors and texture, they are both plausible when considered individually.

On the other hand, SSIM gets worse for all experiments, especially for black pixels. Indeed, luminance, contrast, and structure of the filling used for the user masked area can vary drastically from the rest of the image, and despite all our efforts to prevent the network from considering this region during test-time training, it is still likely influenced by what is used to fill the user masked area. In the case of random pixels, used for the baseline, there is a very wide range of pixel intensities, which diminishes this effect of variation between the valid context and filling. However, since the user mask covers less than one third of the image in our experiments, the carved out portion of a random image or a coarse approximation probably have a very steady luminance, contrast and structure, all of which can be different than those of the valid context of the image, thus exacerbating the variation between the valid context and filling. This is particularly true for black pixels, which explains why it performs the worst.

The L1 score, which increases when color resemblance between the completed region and its ground truth increases, greatly improves for random image and coarse approximation. In fact, the two perform similarly well, which is unexpected. Indeed, as mentioned above, it is reasonable to assume that the filling used to complete the user input affects the network when completing the image. Coarse approximation produces similar colors that can be borrowed by the network, which explains why its L1 score is above the baseline. However, a randomly selected image can be completely different, and consequently might not offer any similar colors to borrow; yet, random image still performs as well as coarse approximation. One hypothesis to explain this would be that we ran this experiment on a very small dataset of 116 samples, and for many of them, we might have inadvertently sampled an image from the same natural scene category, thus increasing the chances of finding similar colors in the filling. On the other hand, with black pixels, the model is limited to borrowing black patches, which explains why it did not perform significantly better than the baseline. Finally, for similar reasons than for L1, PSNR slightly improved.

4.5 Training parameters

In this section, we evaluate our framework when slightly adjusting what it learns and how fast it does it. More precisely, we present the effect of increasing and decreasing the learning rate, turning off the discriminator, freezing earlier layers, and using a background loss.

4.5.1 Learning rate

For all the experiments so far, we set the initial learning rate to the same value as when pretraining the network, which is 1×10^{-6}. Here, we test different initial values for this hyper-parameter, while keeping the division by 2 at every 50 steps. First, we increase it to 2×10^{-6} to see if learning can be achieved more quickly. Then, we decrease it to 1×10^{-7} to see if training becomes more stable when optimizing the gradient in very small increments.

The results are presented in Figure 4.9. Decreasing the learning rate has the expected effect, where the improvement curve is smooth and stable, but the final result is below the baseline. On the other hand, increasing the learning rate makes the model learn faster in the first 20 steps, but then, except for the FID, it becomes almost monotonically decreasing, which implies that overshooting might be occurring. Even for the FID, we notice abrupt spikes in the curve until the halving at epoch 50. We also tried a learning rate of 3×10^{-6}, but this only exacerbated the effects already seen. This possibly means that there is not sufficient additional information to learn, or in other words, that the state of the network is already close to a minimum, either local or global. Even though increasing the learning rate did not work well for this experiment, it might work better when applied jointly with random augmentations, given that they add new examples to learn from. This will be tested in Section 4.6.

4.5.2 Generator only

Our network is divided into two parts: the generator and discriminator. The task of the discriminator, which is to determine whether a given image is real or not, is not an easy one. When trained on a large dataset of examples, the discriminator needs to understand well the semantics of images it is presented at a high and low level in order to make that decision. However, if only given one example, the task becomes much simpler, and the discriminator might learn that looking at only a few values is sufficient to tell if an image is fake or not. For our framework, we primarily want the network to learn how to produce content that can better complete the image at hand, which is the task of the generator. However, since there is only one image, the discriminator might learn very quickly how to make a choice with certainty, which could prevent the generator from learning adequately because of the unbalance between the progression of the two components. Therefore, for this experiment, we turn off the discriminator, and only finetune the generator. In other words, at test time, we only optimize the subset of parameters pertaining to the generator.

The results are presented in Figure 4.9. Finetuning only the generator produces results that are almost identical to the baseline, implying that the discriminator does not necessarily learn more quickly than the generator as anticipated.

4.5.3 Freezing earlier layers

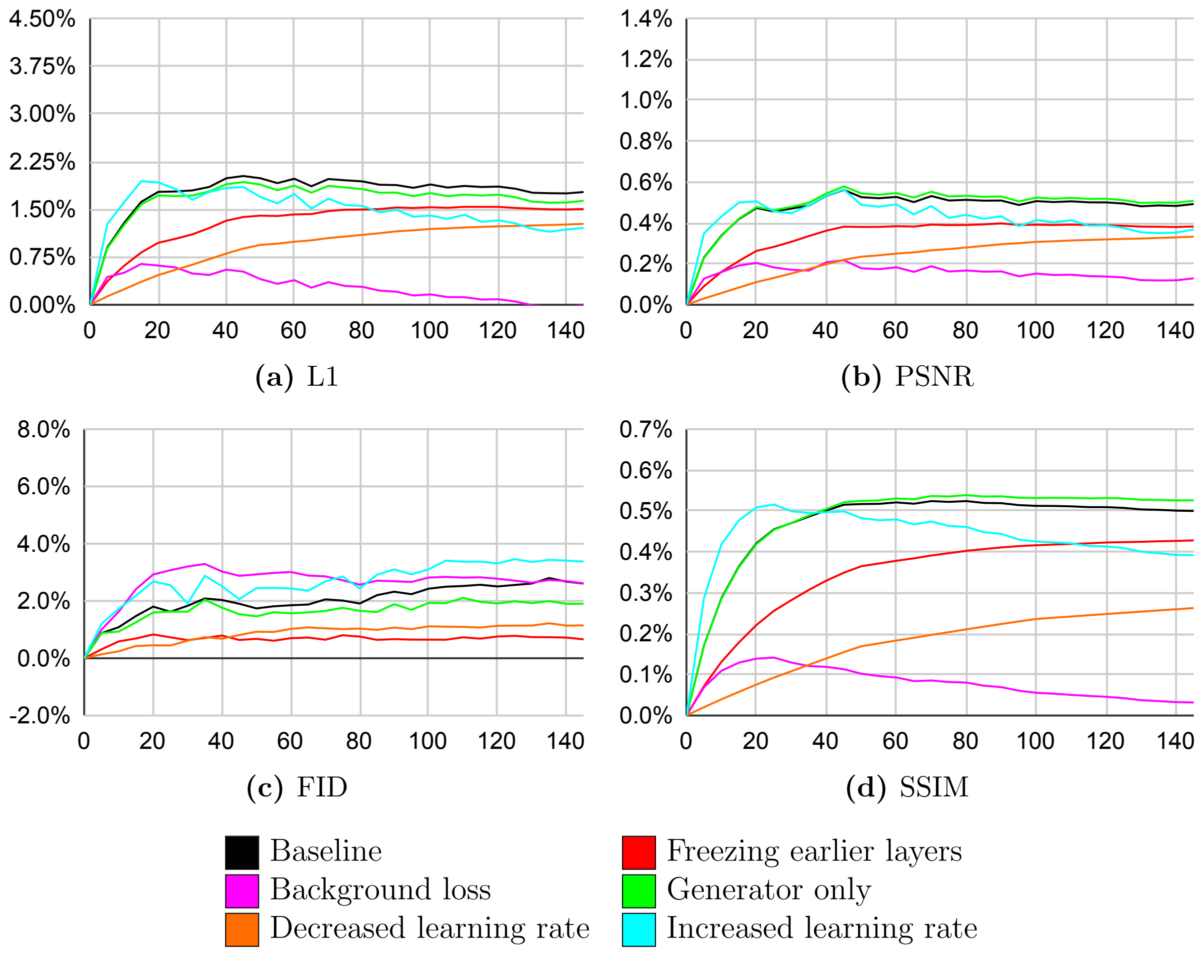
In Section 3.2.3, we mentioned that the last layers are usually more concerned with higher-level semantics specific to each image, whereas earlier layers get an input that is still close to the original image, and consequently they focus on low-level features like colors, texture and structure. These low-level features should usually be handled similarly for all images pertaining to the same distribution. Therefore, in the context of finetuning for a specific image, it is more worthwhile to train later layers than earlier ones. We test this assumption here, by only optimizing the parameters of the generator contained in the layers subsequent to the residual blocks, composed of the three attention layers, the two deconvolutional layers, and the last four convolution layers (see Figure 3.3). For the discriminator, we only optimize the parameters of the last three convolutional layers. This decreases the number of parameters to optimize from 17,849,825 to 7,574,817, thus considerably increasing the speed of test-time training.

Results presented in Figure 4.9 demonstrate a similarity between freezing earlier layers and decreasing the learning rate in that they both produce a much stabler training, but achieve results that are below the baseline. Therefore, the lack of flexibility resulting from freezing almost two thirds of the parameters seems to indicate that it does not let the network reach its full learning potential, and thus, that even adapting the model's interpretation of low-level features can be beneficial in the context of test-time training for inpainting.

4.5.4 Background Loss

Earlier, we described how losses in inpainting are usually applied separately on the foreground and background regions. Here, following the process described in Section 3.2.3, we apply the L1, structural, perceptual, and adversarial losses on both the foreground and background regions, as opposed to only the foreground like is done for all previous experiments.

Aside from the FID, the results in Figure 4.9 show a significant decrease when compared to the baseline. A possible explanation for this is that the network now needs to focus on the background as well, but is still given the same amount of time to learn. Therefore, the attention of the network is distributed among more regions, thus making the output globally more consistent, which is demonstrated by the increase of the FID in the first 50 steps. Nonetheless, the network cannot as freely refine the regions inside the finetuning mask, on which L1, PSNR, and SSIM are computed, because of the additional constraints on the background, and consequently it performs below the baseline for these scores.



(a) L1

(b) PSNR

(c) FID

(d) SSIM

⬛ Baseline
🟪 Background loss
🟧 Decreased learning rate
🟥 Freezing earlier layers
🟩 Generator only
🟦 Increased learning rate

Figure 4.9: Evaluating the results when changing what and how fast the network learns during test-time training. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

4.6 Combinations

In the previous section, we individually applied different settings to our framework and showed how they affected the evaluation scores. In this section, we combine these settings together, and see if their interaction can produce better results. More specifically, we divide this process into three sets. In the first one, we simultaneously apply all random augmentations, and experiment with removing the learning rate adjustment and freezing earlier layers. In the second set, we mainly test different combinations of random augmentations. The third set is built upon the best combination of random augmentations found in the second one, but uses more aggressive learning rates. For all these experiments, we fill the user masked area with a random image as it performed the best in comparison to other types of filling (see Section 4.4), and apply each random augmentation as is done in Section 4.3. Finally, we select the optimal settings that will be used to run the full evaluation, and evaluate how relevant learning on prior examples is for our framework.

4.6.1 First set

Combination A

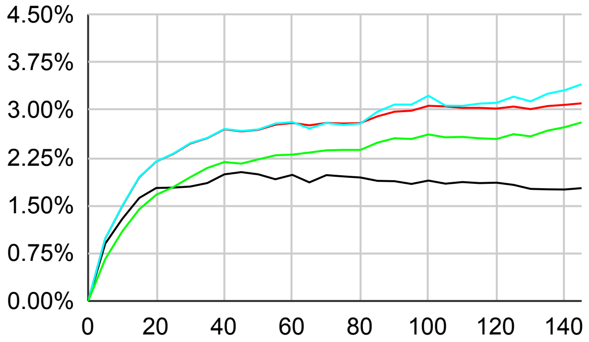
Here, we apply many random augmentations together to see if our framework can learn better when the set of examples it is trained on is more varied. Therefore, we randomly augment the input with a 60% probability, in which case we apply fixed Gaussian blur (with a kernel size of 5 and $\sigma = 2$) 30% of the time, random crop 40% of the time, left-right or up-down flips 50% of the time, and random shear or random rotation with equal likelihood 40% of the time. Moreover, when one of the two flips, either left-right or up-down, is applied, then the other is also applied with a 40% chance. We never apply random shear and random rotation at the same time to avoid creating an example that is too different from the original input.

Combination B

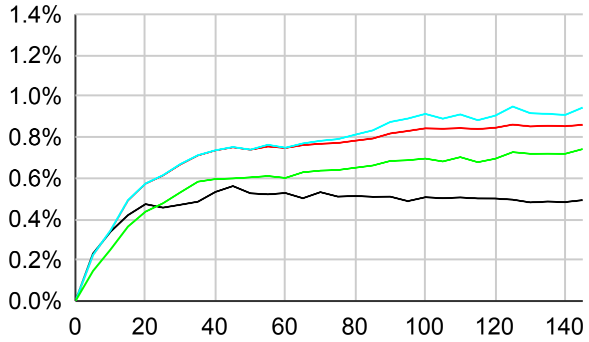
For this combination, we test the effect of a constant learning rate when applying random augmentations to the input. The rationale is that more examples generate additional cases to learn, which means that overshooting is less likely to occur, and it should be safer to use the same initial learning rate without reducing it even in the later steps. Therefore, we apply exactly the same random augmentations with the same probabilities as for combination A, and remove the division by 2 of the learning rate at every 50 steps.

Combination C

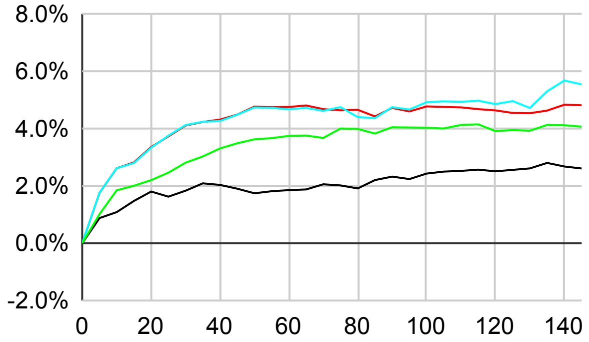
For combination C, we start with the same settings used for combination B, and freeze earlier layers like is done in Section 4.5.3, with the exception that we also freeze the third convolutional layer of the discriminator, and only freeze the first four residual blocks of the generator, as opposed to all eight, making the number of parameters go down from 17,849,825 to 11,775,009. With this, we hope to see if random augmentations can make our framework perform above the baseline, even when only optimizing the later layers of the network.



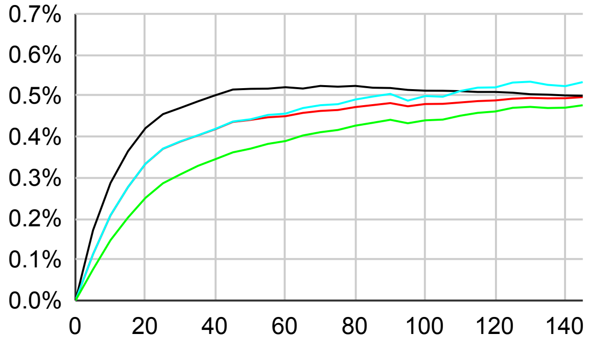
(a) L1



(b) PSNR



(c) FID



(d) SSIM

⬛ Baseline 🟥 Combination A
🟦 Combination B 🟩 Combination C

Figure 4.10: Evaluation of the first set of combinations. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

Results

The results for this first set of combinations are presented in Figure 4.10. All combinations of the first set outperform the baseline, with the exception of combination C that achieves slightly inferior results for SSIM. This means that providing our framework with more examples has the desired effect of making it learn more. As expected, combination B performs slightly above combination A, which means that additional learning can be made in later steps, and increasing the learning rate even at the beginning might be beneficial. Finally, combination C produces a slightly less smooth curve than Experiment 4.5.3, which is expected as fewer parameters are frozen. Moreover, even with a substantial decrease in the number of parameters, it still performs similarly to combinations A and B, which confirms that most of the learning tailored to a specific image happens in later layers. Finally, unlike for Experiment 4.5.3, combination C produces results above the baseline, thus suggesting that the network takes advantage of random augmentations.

4.6.2 Second set

Combination D

For this combination, we increase the number of steps from 150 to 300, and only adjust the learning rate at step 150 by halving it. We also randomly augment the input with a 65% chance, in which case we apply random brightness 50% of the time, random Gaussian blur 40% of the time, random crop 40% of the time, left-right flip 50% of the time, and random shear 40% of the time. The objective of this experiment is to see if, when paired with random augmentations, our framework can continue learning for longer. Moreover, we only apply augmentations that keep the image in an upright position. Indeed, in Section 4.3.2, we found out that the network is very sensitive to rotations of the image, given that natural scenes often offer the same ordering of colors from top to bottom, and therefore, we try to mitigate this effect.

Combination E

For combination E, we keep the same settings as for combination D, but apply the random augmentations like is done for combination A. The goal is to validate that augmentations changing the orientation of the image, such as random rotation and up-down flip, indeed affect the ability of the network to understand a natural scene. This validation is done by comparing combination E to combination D, which doesn't include such augmentations.

This experiment also allows us to see if combination B can produce better results when trained for longer.

Combination F

We again keep the same settings as for combination D, but include random translation in the following way. If random crop is not applied (60% chance), then random translation is applied instead with a probability of 80%. This adds one more possible augmentation, and allows us to see how random translation interacts with all the other transformations. We do not apply random crop and random translation simultaneously, since the former already only retains 25% of the information, and with random translation, this percentage would drop even further.

Combination G

So far, our process has been to generate a finetuning mask, and to augment it with the same random augmentations as those applied to the image, except for random brightness, random Gaussian noise, and random Gaussian blur. This increases the diversity of finetuning masks, and also allows them to follow the transformation of the image, such that they do not end up covering empty pixels (see Section 4.3.2). However, we found out that the resulting finetuning masks have shapes that are not always well recognized by the network, and also that, in the case of random crop or random shear, they are sometimes too small. As mentioned in Section 4.2, inpainting networks are very sensitive to the shapes and sizes of masks. Therefore, for this experiment, we use the same settings as for combination F, but we do not augment the finetuning mask, which allows us to evaluate the impact on the results of augmenting the mask in the same manner as the image.

Since the finetuning mask no longer follows the transformation of the image, we need to include an additional constraint to decide if a sampled mask is retained or not (the current process is described in Section 3.2.2). Indeed, for random augmentations that create empty pixels, like random shear, it is undesirable to have the finetuning mask cover these pixels. Therefore, we carve out the portions of the finetuning mask that overlap with empty pixels, and make sure that it still covers at least 10% of the image after this operation; if it does not, then we sample a new one.

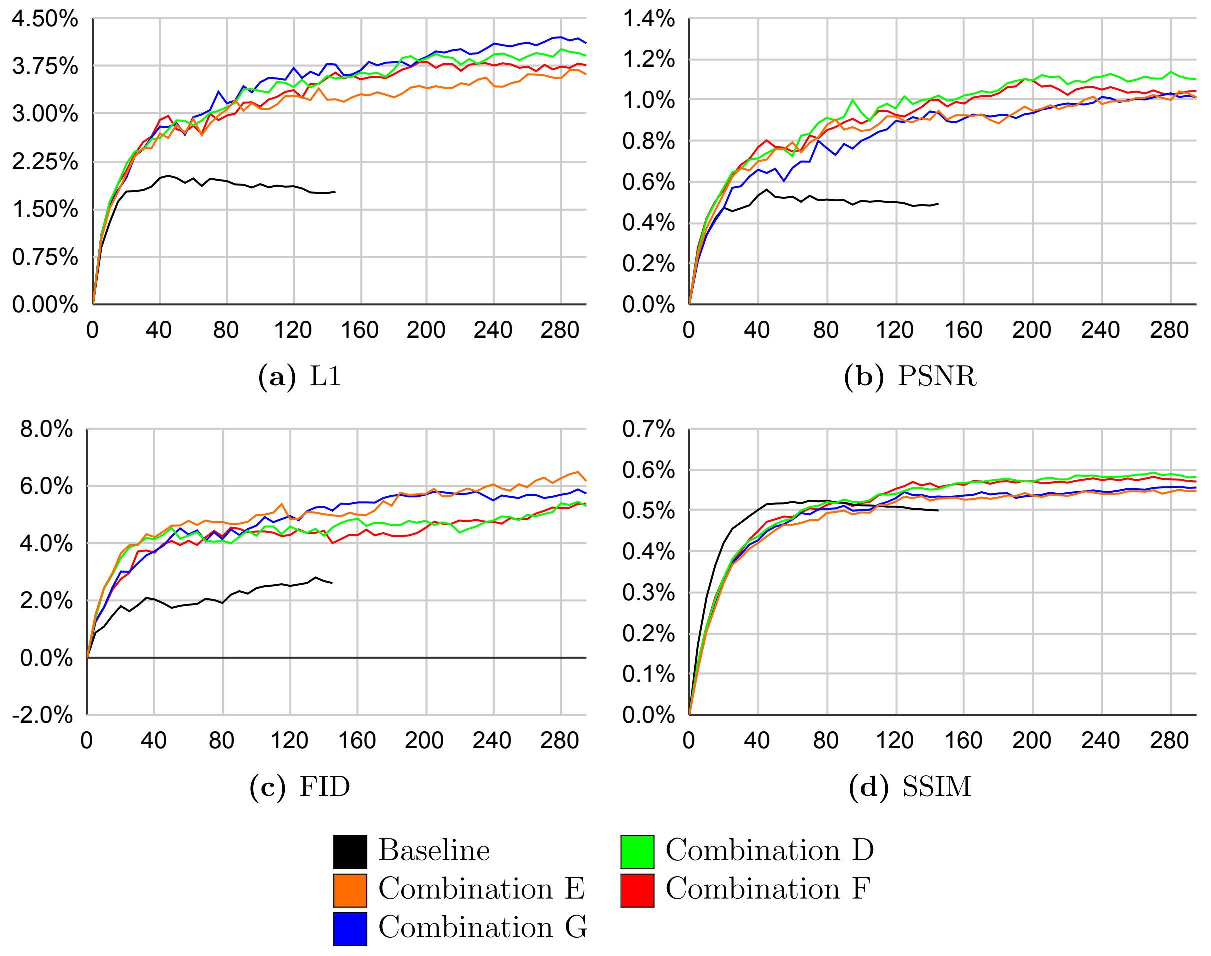


Figure 4.11: Evaluation of the second set of combinations. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

Results

The results for the combinations of the second set, presented in Figure 4.11, are all very similar, and well above the baseline. Other than for the FID, combination E outperforms combination D, but it is not significant enough to confirm our hypothesis that augmentations changing the image orientation have a negative impact on the result. Results for combination F are all slightly below those of combination D, which suggests that random translation is not beneficial when applied in combination with the other augmentations used. As for combination G, it performs relatively well, but again, the results are not significant enough to support our assumption that augmenting the mask makes it less recognizable by the network and is detrimental for the results.

Finally, even though the increase is less substantial than during the first 150 steps, the last 150 steps still show improvements, which means that the network is not done learning at step 150. Nonetheless, it doubles computation time, which is why, in the next set, we increase the learning rate to see if this additional learning can be achieved faster.

4.6.3 Third set

Combination H

Here, we use the same settings as for combination G, but do not apply random translation. We make this change because combinations D and F demonstrated that adding random translation to the set of augmentations applied on the input produces a decrease in the results. Therefore, the goal is to evaluate the impact on the results of augmenting the mask, when random translation is not used.

Combination I

For combination I, we use the same settings as for combination H, but increase the initial learning rate to 2×10^{-6}. We also reduce the number of steps to 150, and do not use any learning rate adjustment. The objective is to see if learning can be achieved faster.

Combination J

Combination J is exactly the same as combination I, but it uses a higher initial learning rate of 3×10^{-6} to see how fast can our framework learn before we start noticing overshooting.

Results

Results in Figure 4.12 confirm that learning can be achieved faster, but at the cost of more spikes in the evolution of scores over time, suggesting that overshooting starts to happen. While combinations I and J produce very similar results, the former finishes slightly higher (see Table 4.1), and contains smaller spikes for the FID metric. Finally, combination H is slightly above combination G, but it is not significant enough to confirm that random translation has a negative impact. Nonetheless, since the gap between H and G is slightly smaller than the gap between F and D, it could mean that not augmenting the finetuning mask makes the network more robust to additional random augmentations, because there is less diversity in the finetuning masks, and thus the network can focus more on adapting to the image.

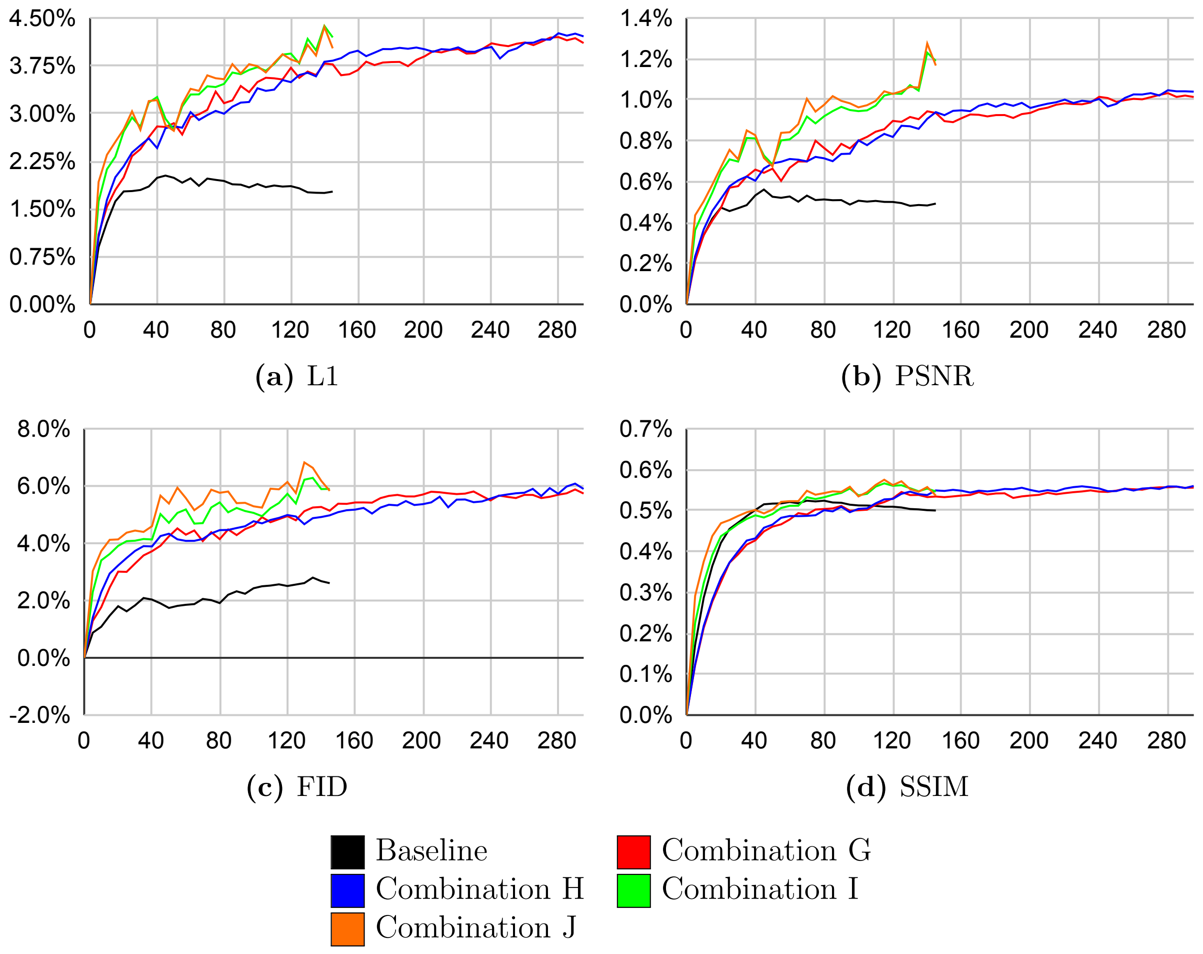


Figure 4.12: Evaluation of the third set of combinations. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

4.6.4 Optimal settings

Now that we have evaluated each setting individually, and also their interaction when combined, we are ready to select the settings that will be used to run our final evaluation in Section 4.7. We base our decision on the evaluation metrics, i.e., L1, PSNR, SSIM, and FID, but also on computation time. This is represented in Table 4.1, where better results are at the top, and where purple highlighting indicates experiments that are run for 300 steps, while others are only run for 150 steps. Among the combinations trained for 300 steps, there is not a clear winner, since H, D, and E perform the best depending on the metric. Moreover, training for 300 steps is computationally expensive. On the other hand, among the experiments trained for 150 steps, it is easy to see that the best performing combination is I. Indeed, its difference with the first contender for L1 is negligible, it performs the best for PSNR, and is in second place for FID, with the first place being occupied by an experiment run for 300 steps. As for SSIM, the results are all very close, and except for combinations A and C, these differences are not significant. Therefore, since combination I is only trained for 150 steps and outperforms or is close to outperforming all of the other combinations, even those trained for 300 steps, we select its settings to run the full evaluation. In the next sections, we will refer to the settings of combination I as the optimal settings.

L1	PSNR	SSIM	FID
H (4.21%)	I (1.19%)	D (0.58%)	E (6.17%)
I (4.20%)	J (1.17%)	F (0.57%)	H (5.90%)
G (4.11%)	D (1.10%)	G (0.56%)	I (5.90%)
J (4.02%)	F (1.04%)	H (0.56%)	J (5.83%)
D (3.91%)	H (1.04%)	E (0.55%)	G (5.74%)
F (3.76%)	E (1.01%)	I (0.55%)	B (5.54%)
E (3.62%)	G (1.01%)	J (0.54%)	F (5.39%)
B (3.40%)	B (0.94%)	B (0.53%)	D (5.30%)
A (3.10%)	A (0.86%)	A (0.50%)	A (4.81%)
C (2.80%)	C (0.74%)	C (0.48%)	C (4.07%)

Table 4.1: Evaluation metrics of all the tested combinations, with the ones achieving the best results at the top. Experiments highlighted in purple are run for 300 steps, while the others with no color are run for 150 steps.

4.6.5 Zero-shot learning

Before running the full evaluation, there is one more assumption that needs to be tested. So far, we have assumed that prior knowledge gained from looking at other images could be helpful when completing a new, unseen image. Consequently, we first pretrained a model on a large set of known examples before using it as a starter point for our framework when completing user masked inputs. In this section, inspired from [24], we apply our framework without relying on a pretrained network. This technique is commonly referred to as “zero-shot learning”. In our case, the objective is to evaluate how relevant is learning from other examples prior to completing user inputs. In other words, given that our framework is designed to handle a single image, we are wondering if the valid context of the input at hand alone is sufficient to adequately complete the image. If it were the case, our framework could be generalized to a lot more images. Indeed, one constraint of our approach is that in order to obtain optimal results, the user input should be pulled from the same distribution as the examples of the dataset on which the network was pretrained. As mentioned before, gathering data is a long and hard task, which means that the type of user inputs is limited to the image distributions available. It would also remove the need of pretraining a network on each of these distributions, which requires a tremendous amount of time.

To conduct this experiment, we use the optimal settings described in Section 4.6.4, except for the learning rate, which we increase to 5×10^{-6}. We first use the same network architecture, and apply our framework by randomly initializing its parameters as opposed to restoring them to the state of the pretrained model. Then, we also evaluate zero-shot learning by using a significantly smaller network architecture. The rationale is that the more examples there are, the more parameters are needed to appropriately model all cases that are seen. However, here we only need to learn how to fill one image, which greatly simplifies the modelling task as there are far fewer cases to adapt to. We therefore only use a small subset of the original architecture by removing 6 of the residual learning blocks, and only downscaling the images to 128×128 as opposed to 64×64, which requires one fewer downsampling and upsampling layer. Consequently, the total number of parameters for this second part of the experiment decreases from 17,849,825 to 2,022,192.

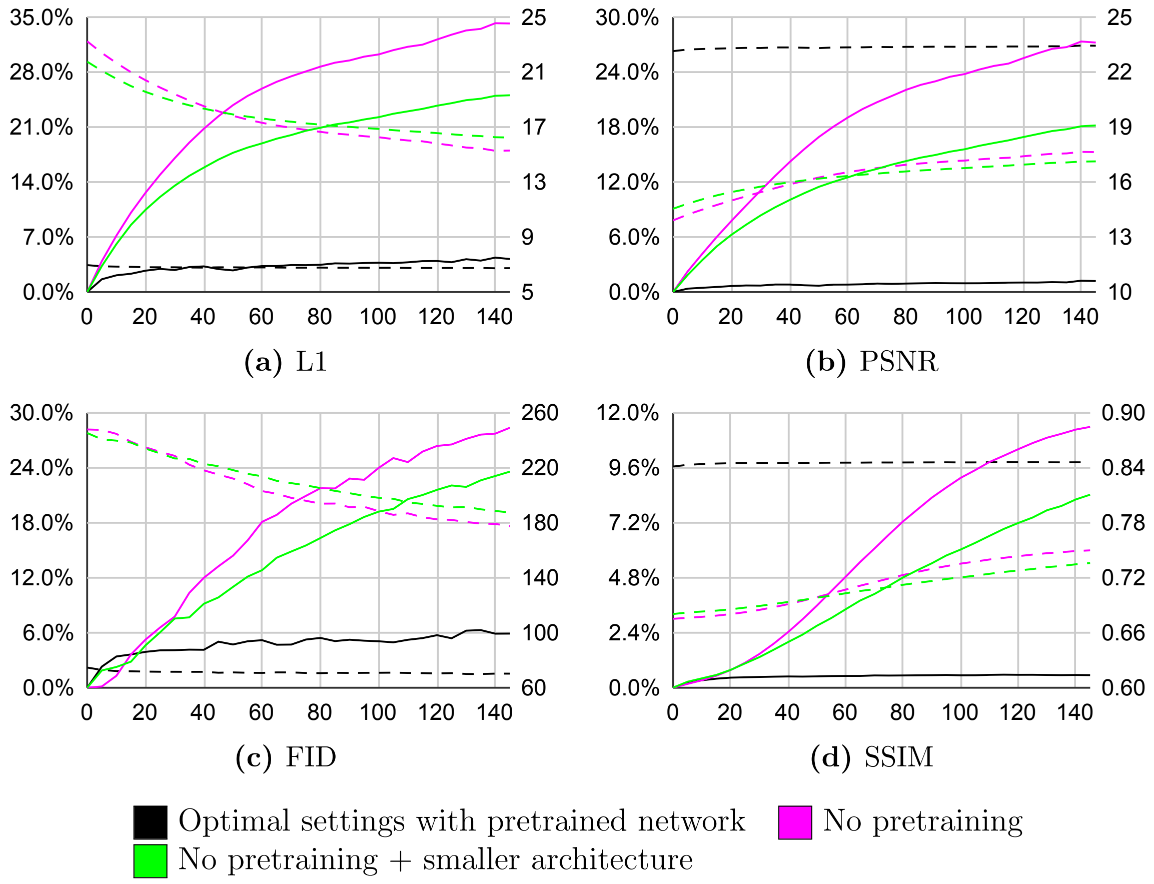


Figure 4.13: Evaluation of zero-shot learning. On the x-axis is the finetuning step. The solid lines depict the improvement over the original score, in percentage, and are linked to the **left** y-axis. The dashed lines represent the average value of the score, and are associated with the **right** y-axis. The average is taken over all 116 images processed for this experiment. For L1 and FID, lower scores are better, whereas it is the opposite for PSNR and SSIM.

The results of this experiment are shown in Figure 4.13. We observe that the improvement over the original scores is very high, and much above the improvement obtained with the optimal settings using a pretrained network. This is expected as the original score is based on a purely random completion of the image as the network is no longer pretrained, which gives more room for improvement. In other words, when initialized, the network is nowhere close to a minimum, which makes the optimization task easy at the beginning, and thus produces larger improvements. However, when looking at the dashed lines, we can see that the evaluation scores are much worse for the zero-shot experiments than for optimal settings using a pretrained network, which means that even though the improvement is considerable, it is still not able to reach the accuracy of the pretrained network. To handle this, we could always use a more aggressive learning rate, or finetune for more steps, but Figure 4.13 shows that the derivative of the slopes at step 150 is close to 0, meaning that it will probably soon reach a plateau, and thus that there is likely not much additional learning that can be done. Another important observation is the proximity of the curves for the two zero-shot experiments, which suggests that zero-shot learning with a smaller architecture does not perform much worse than using the full size architecture. Therefore, this supports our assumption that the network does not need as many parameters when only adapting to a single image. Finally, visual results show that the completed user masked region is almost exclusively random pixels, which concurs with the results and allows us to conclude that prior knowledge is relevant, and that the valid context of the user input, alone, is not sufficient to adequately complete the missing regions.

4.7 Full evaluation

So far, to efficiently test many different settings, we have only applied our framework to a small set of 116 samples. In this section, we apply our framework to a much larger dataset of 10,000 images randomly selected from Places2 validation dataset. The objective is to confirm that the results obtained heretofore are reproducible at a larger scale, and also that our framework generalizes well to other natural scene categories. To execute this full evaluation, we use the optimal settings defined in Section 4.6.4.

4.7.1 Results

The results of this full evaluation are presented in Figure 4.14. One noticeable effect of applying our framework on 10,000 images is that it creates a much stabler improvement over time, demonstrated by the smoothness of the learning curves. This is expected, as the more data points there are, the less impact outliers have, which creates an average that is more consistent from step to step. Another side effect of this increase in the number of images evaluated is the substantial improvement of the FID. This is very important as perceptual scores are widely esteemed for evaluating inpainting networks. The fact that our framework is able to achieve an improvement of over 15% is not negligible and shows the potential it has. Moreover, using more samples produces a much more robust evaluation of the true value of the FID for our framework. Indeed, as previously explained in Section 4.3.2, the variance and covariance between all existing image features in the dataset are directly proportional to the FID. In statistics, it is well known that a smaller sample is very likely to have more variance than a larger one. Therefore, using a large set of 10,000 images to evaluate the FID reduces the sample variance and covariance, which improves the score (the lower the FID is, the better) and gives a better approximation of the improvement our framework is capable of achieving.

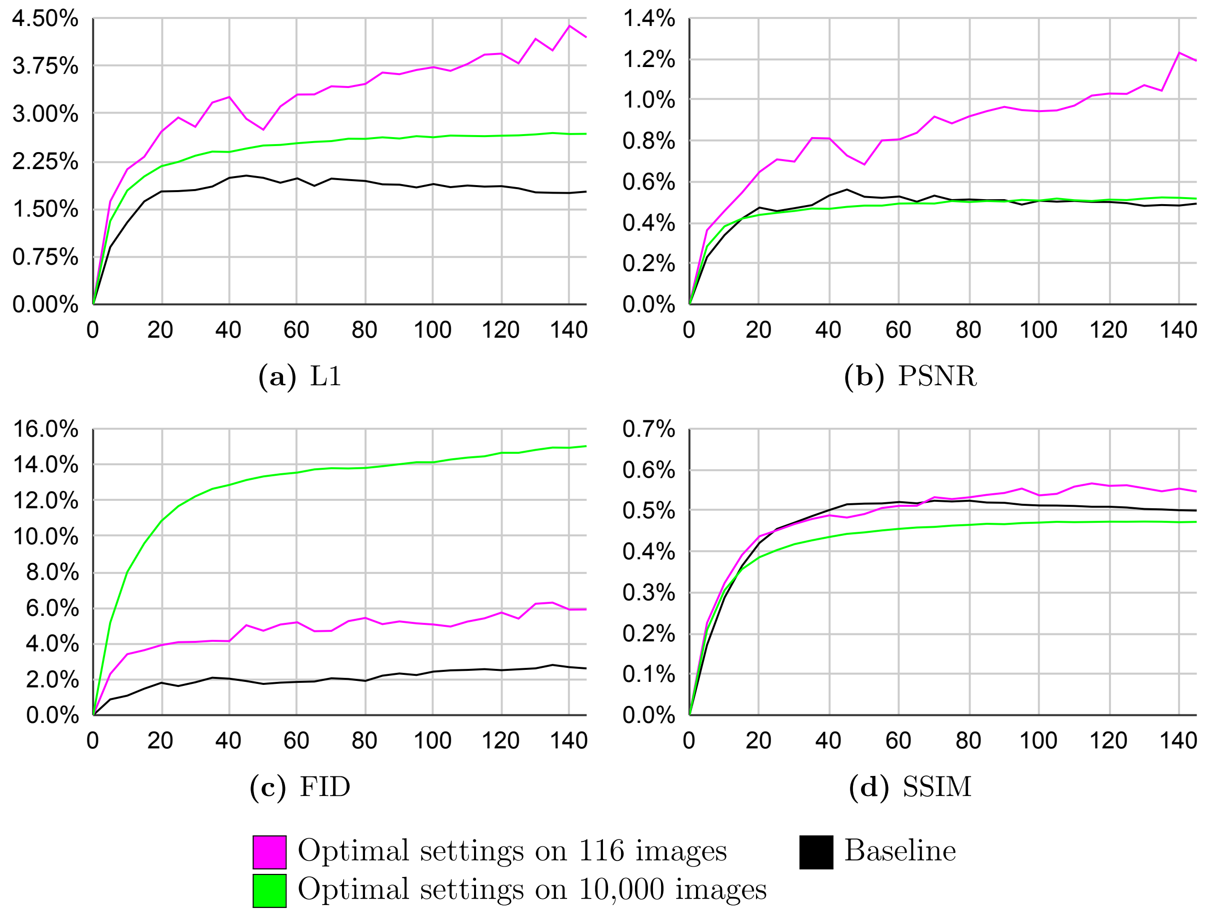


Figure 4.14: Evaluation of the optimal settings on 10,000 images. On the x-axis is the finetuning step, while on the y-axis is the improvement over the original scores, in percentage.

Even though the full evaluation leads to a much stabler improvement over time, it produces results that are not as good as the initial experiment on 116 images for L1 and PSNR. Nonetheless, L1 still performs significantly better than the baseline, while PSNR is slightly above, which means that the results are satisfactory for these two metrics. In fact, the original experiment on 116 images produced results above expectation. We explained in Section 4.4 that it was unexpected for filling the user masked area with a random image to produce similar results as filling it with a coarse approximation. This is because a patch extracted from a purely random image can have completely different colors than the valid context of the input, whereas a coarse approximation will always feature similar colors. Nonetheless, we attributed this success to the low number of different categories of natural scenes present in the dataset at test time, which consequently increased the likelihood of filling user holes with an image from the same category having similar colors. However, with 10,000 images, this coincidence is a lot less likely to occur, which is why the results for L1 and PSNR, presented in Figure 4.14, are expected and satisfactory even though they are not above those of the original experiment.

Finally, SSIM does not vary a lot from the baseline or the original experiment, and does not seem to be a good indicator of the overall performance of our framework. Given that the learning curves of all evaluation metrics are almost monotonically increasing, we can conclude that our framework is stable, and that it can lead to improvements even at a larger scale.

4.7.2 Individual improvements distribution

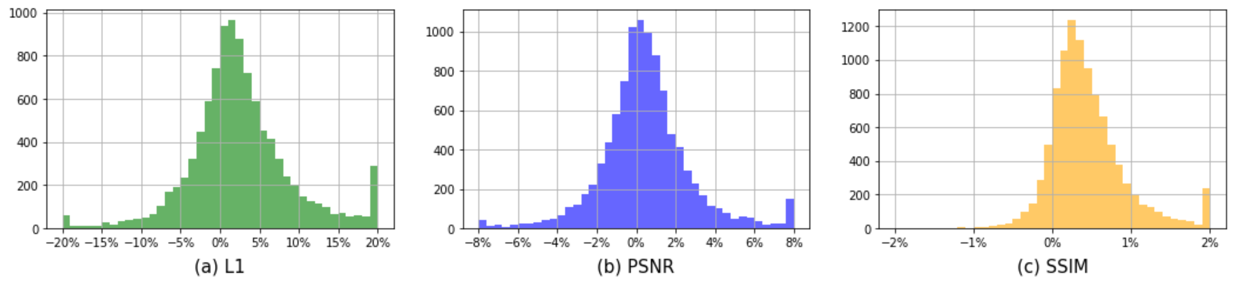


Figure 4.15: Distribution of evaluation scores at step 150 for L1, PSNR, and SSIM. The x-axis represents the number of samples in each bin of the histogram, whereas the y-axis represents the final improvement after 150 steps of test-time training. The sudden increases in the number of samples at both extremities of each histogram represent the outliers that were clipped to fit in the figure.

The results presented in Section 4.7.1 are obtained from taking the average score of the 10,000 samples at different steps during finetuning. However, considering just the average does not give any insight into how each sample performed. In this section, we look at the distribution of these scores at step 150 for additional clues, such as outliers skewing the average, or even patterns indicating possible improvements for our framework. This can only be done for L1, PSNR and SSIM, as FID is a sample score, and thus cannot be computed for a single data point.

Figure 4.15 shows the distribution of individual scores at step 150 for L1, PSNR and SSIM. We can observe that, while our framework does not perform well for some images, it performs well for many other images. In addition, all the distributions are slightly shifted to the right of 0%, which means that there are more images that get improved than the contrary. The fact that our framework performs very well in some cases makes it a great asset to integrate into inpainting software. With it, the user could choose from the raw and finetuned output. This way, if the latter is worse or does not present any major improvement, then the user can decide to keep the raw output. Used like this, our framework could only be beneficial.

4.7.3 Visual results

In this section, we exhibit a few visual results that clearly demonstrate the types of improvement our framework is capable of. These results are obtained using the optimal settings described in Section 4.6.4, and are achieved on pictures taken from Places2 validation dataset, which have not been used for pretraining the network.

The visual results are presented in Figure 4.16, where the raw and finetuned outputs are put side by side. In the first row, the waves show how our framework can bring more structural consistency and reduce the amount of undesirable artefacts. In the second row, the side wall of the barn presents less color bleeding from the snow when completed with our framework. In the third row, the color consistency of the yellow region of the field is much better in the finetuned result. Moreover, there is also a more natural flow of colors in the trees area. Finally, the cliff presented in row four demonstrates a better structural completion of the forest, and also a reduction in artefacts and color bleeding when completed with our framework. Additional results are also presented in Figures 4.17 and 4.18. We also include in Figure 4.19 a few examples for which our framework did not produce an improvement.

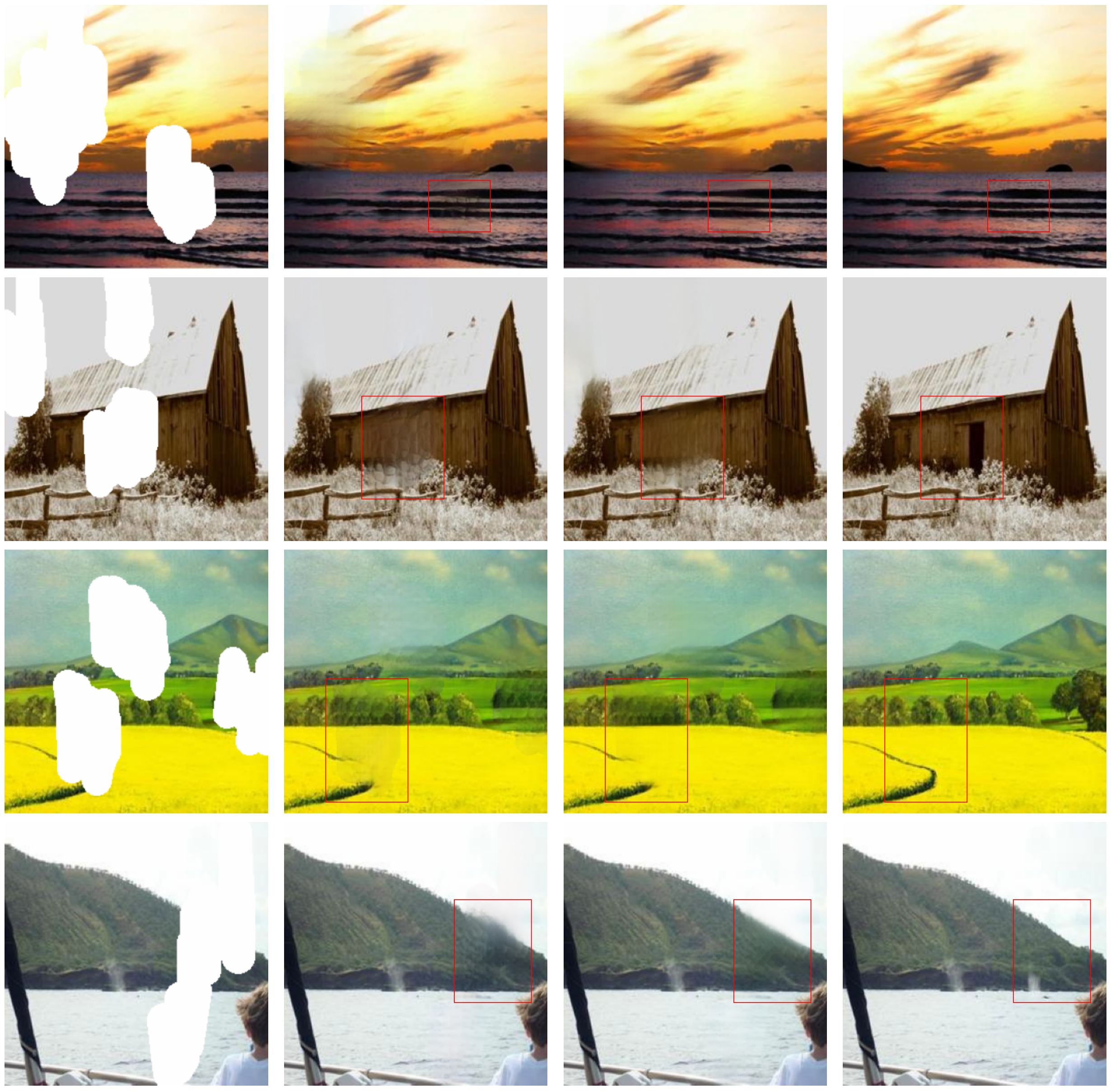


Figure 4.16: Visual results obtained with optimal settings. All of the images are taken from Places2 validation dataset. The red bounding boxes indicate important areas of improvement. "Raw" designates the pretrained network without any finetuning, whereas "Ours" is for our finetuning framework.

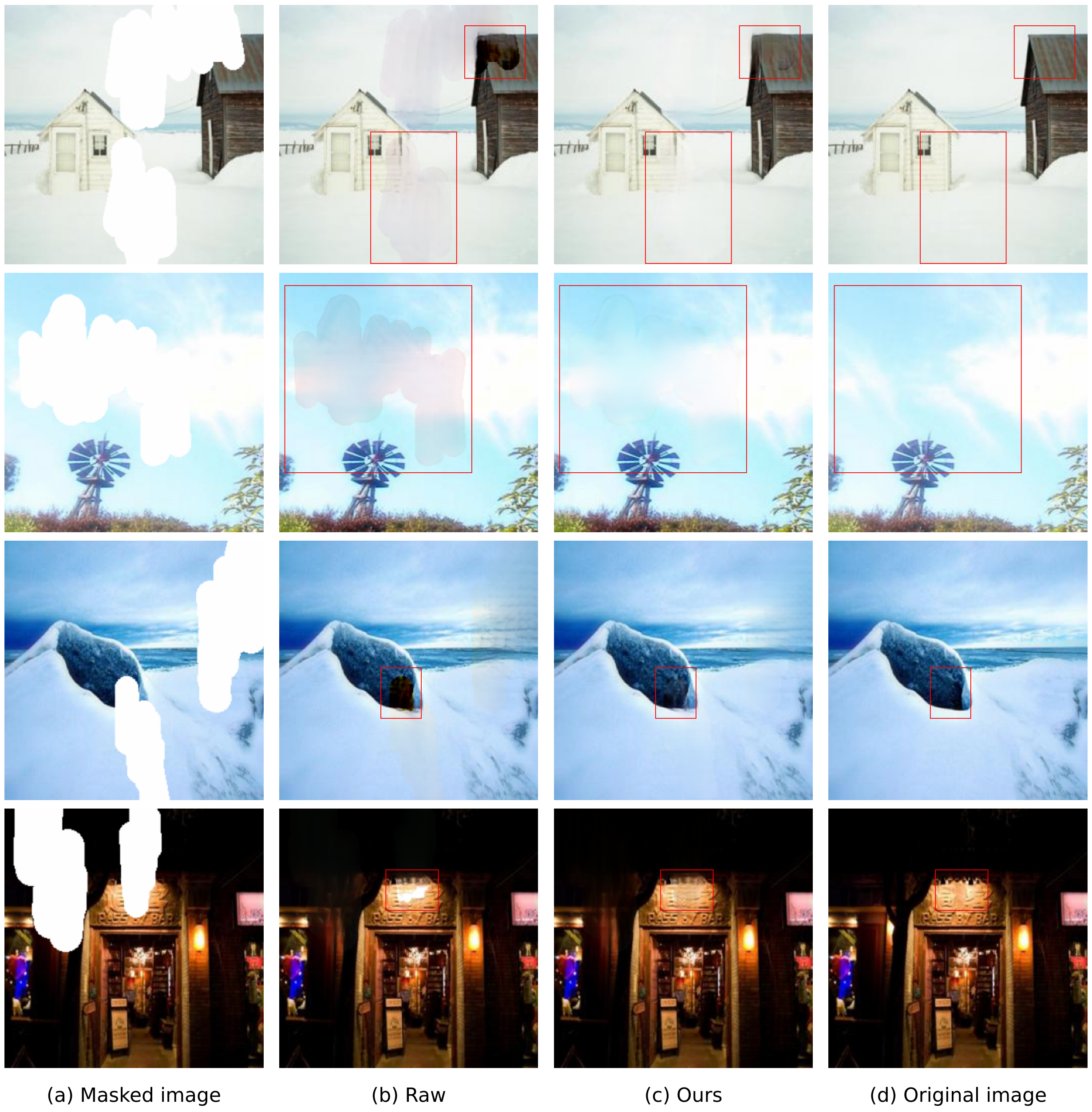


Figure 4.17: More results from our framework on images taken from the validation dataset of Places2 (1).

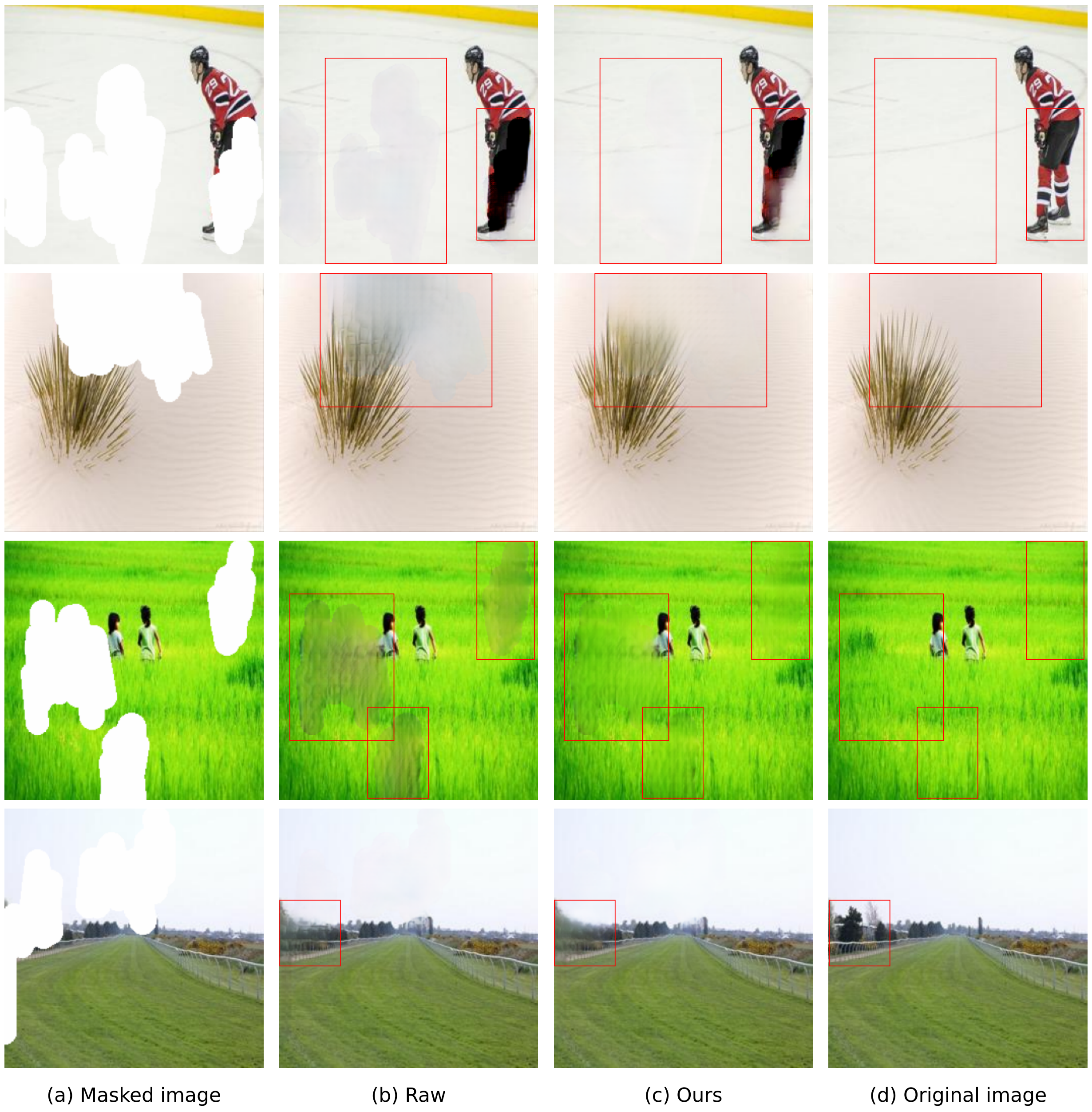


Figure 4.18: More results from our framework on images taken from the validation dataset of Places2 (2).

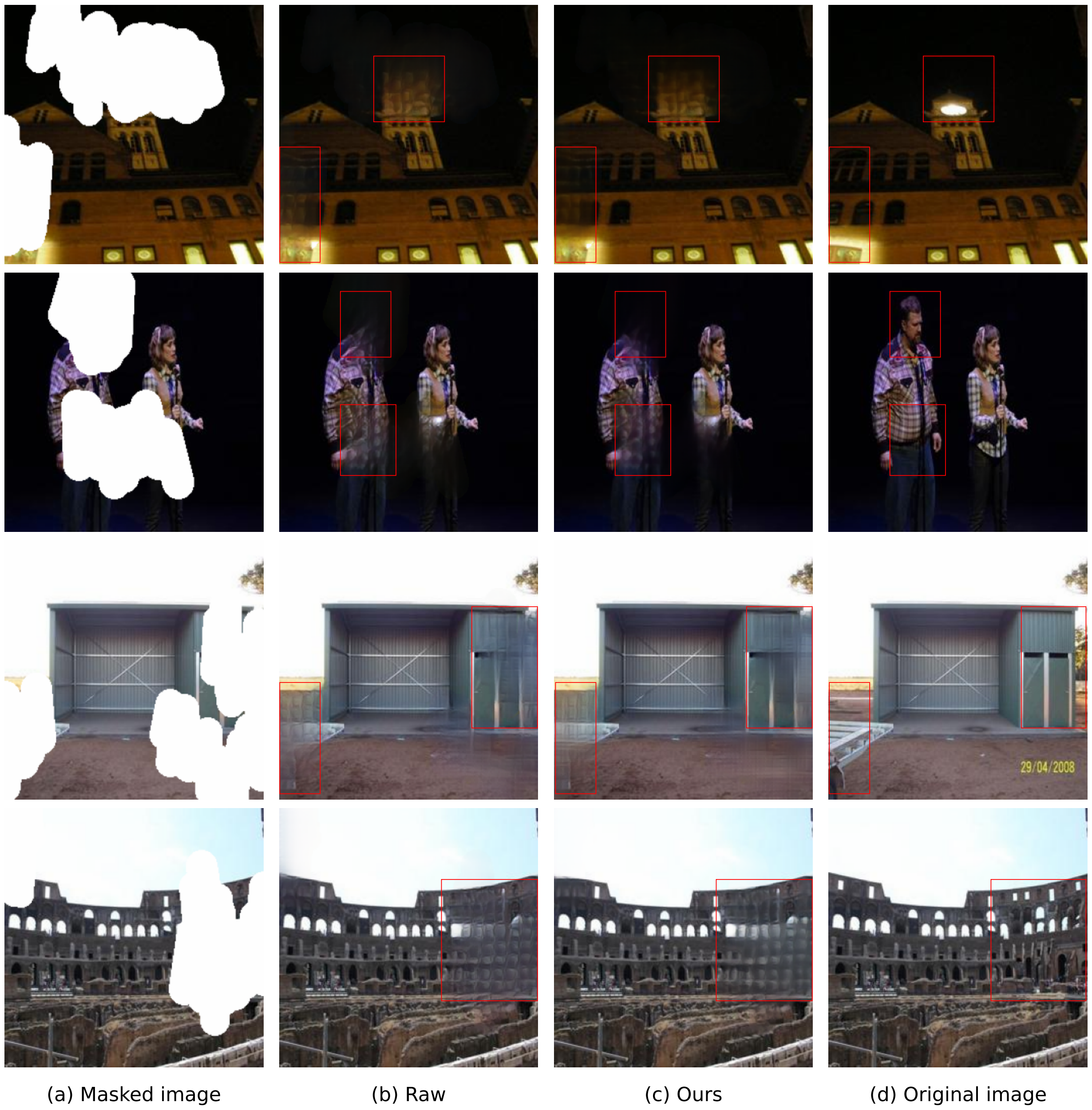


Figure 4.19: Visual results for which our framework did not produce an improvement. Images are taken from the validation dataset of Places2.

Chapter 5

Conclusion and Future Work

Image inpainting is the task of completing missing regions in images. Traditionally, a CNN is trained on a large dataset of examples for many iterations before it can be used for inference on new, unseen images. However, each image belongs to a certain domain, and in the image space, these domains are extremely varied. Therefore, this adds the constraint that, for optimal results, the image used at inference time should belong to the same data distribution as the examples of the training dataset. In this thesis, we propose a test-time learning approach for image inpainting. More precisely, we train a CNN for this task on a large dataset as usual, but use the user image containing holes to generate a new dataset on which we further finetune the CNN. With this, we hope to optimize the network such that it learns to fill holes specific to the user image better, and also to allow a wider range of image domains at test time to produce optimal results. When evaluated on 10,000 images, taken from the validation dataset of Places2, with the most optimal settings found, our method shows positive improvements on both visual results and evaluation metrics. This is especially true for FID, where an improvement of more than 15% in comparison to the original score is achieved. Moreover, this framework can easily be added to most existing networks without any real drawback except for a longer computation time, since the user can always select which image they prefer between the original one and the finetuned one.

In future work, we first plan on testing different backbone models to confirm that our framework is compatible and that our results are reproducible. Second, we plan on testing different architectures on top of the pretrained network. Indeed, in this work, we limited ourselves to optimizing the parameters of the pretrained network during test-time training. However, our framework might produce further improvements if we design a structure on top of the network tailored to this finetuning task, and only optimize the parameters of this smaller structure. With this, not only would we reduce computation time because there would be less parameters to optimize, but we would also have the flexibility of adding our own layers as opposed to being obligated to reuse the ones of the backbone model. Third, we found that using random images to fill user holes produced the best results. However, because the image is randomly selected, it might be completely different than the valid context of the user input, which might explain why our framework performed poorly in some cases (figure 4.15). It would therefore be interesting to see if these results improve when computing image similarities, and filling the user masked area with an image that has similar colors, texture, and structure to those of the valid context. Fourth, a potentially interesting idea would be to train a CNN by manually generating new composited images in which a foreground object is pasted onto a background image, and then used as an object to remove via inpainting. This would work well because the ground truth is available. Finally, it would be interesting to exploit EXIF data in user images to find "companion" images that were captured at roughly the same time and location. These "companion" images would likely be similar to the user image, and could be used to complement the dataset during test-time training.

References

[1] Marcelo Bertalmio, Guillermo Sapiro, Vincent Caselles, and Coloma Ballester. Image inpainting. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 417–424, 2000.

[2] Vincent Casser, Soeren Pirk, Reza Mahjourian, and Anelia Angelova. Depth prediction without the sensors: Leveraging structure for unsupervised learning from monocular videos. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 8001–8008, 2019.

[3] Yuhua Chen, Cordelia Schmid, and Cristian Sminchisescu. Self-supervised learning with geometric constraints in monocular video: Connecting flow, depth, and camera. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7063–7072, 2019.

[4] Ramazan Gokberk Cinbis, Jakob Verbeek, and Cordelia Schmid. Unsupervised metric learning for face identification in tv video. In *2011 International Conference on Computer Vision*, pages 1559–1566. IEEE, 2011.

[5] Antonio Criminisi, Patrick Perez, and Kentaro Toyama. Object removal by exemplar-based inpainting. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages II–II. IEEE, 2003.

[6] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[7] Omar Elharrouss, Noor Almaadeed, Somaya Al-Maadeed, and Younes Akbari. Image inpainting: A review. *Neural Processing Letters*, 51(2):2007–2028, 2020.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[9] James Hays and Alexei A Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics (ToG)*, 26(3):4–es, 2007.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[12] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Globally and locally consistent image completion. *ACM Transactions on Graphics (ToG)*, 36(4):1–14, 2017.

[13] Vidit Jain and Erik Learned-Miller. Online domain adaptation of a pre-trained cascade of classifiers. In *CVPR 2011*, pages 577–584. IEEE, 2011.

[14] JK Rahul Jayawardana and T Sameera Bandaranayake. Analysis of optimizing neural networks and artificial intelligent models for guidance, control, and navigation systems. *International Research Journal of Modernization in Engineering, Technology and Science*, 3(3):743–759, 2021.

[15] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[16] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1409–1422, 2011.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[18] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[19] 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)*, pages 85–100, 2018.

[20] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[21] Xuan Luo, Jia-Bin Huang, Richard Szeliski, Kevin Matzen, and Johannes Kopf. Consistent video depth estimation. *ACM Transactions on Graphics (ToG)*, 39(4):71–1, 2020.

[22] F. Perazzi, J. Pont-Tuset, B. McWilliams, L. Van Gool, M. Gross, and A. Sorkine-Hornung. A benchmark dataset and evaluation methodology for video object segmentation. In *Computer Vision and Pattern Recognition*, 2016.

[23] David A Ross, Jongwoo Lim, Ruei-Sung Lin, and Ming-Hsuan Yang. Incremental learning for robust visual tracking. *International journal of computer vision*, 77(1):125–141, 2008.

[24] Assaf Shocher, Nadav Cohen, and Michal Irani. "zero-shot" super-resolution using deep internal learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3118–3126, 2018.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[27] Kevin Tang, Vignesh Ramanathan, Li Fei-Fei, and Daphne Koller. Shifting weights: Adapting object detectors from image to video. *Advances in Neural Information Processing Systems*, 25, 2012.

[28] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9446–9454, 2018.

[29] Gourav Wadhwa, Abhinav Dhall, Subrahmanyam Murala, and Usman Tariq. Hyper-realistic image inpainting with hypergraphs. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 3912–3921, 2021.

[30] Ziyu Wan, Jingbo Zhang, Dongdong Chen, and Jing Liao. High-fidelity pluralistic image completion with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4692–4701, 2021.

[31] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8798–8807, 2018.

[32] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[33] Jie Yang, Zhiquan Qi, and Yong Shi. Learning to incorporate structure knowledge for image inpainting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12605–12612, 2020.

[34] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Generative image inpainting with contextual attention. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5505–5514, 2018.

[35] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Free-form image inpainting with gated convolution. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4471–4480, 2019.

[36] Shun Zhang, Jia-Bin Huang, Jongwoo Lim, Yihong Gong, Jinjun Wang, Narendra Ahuja, and Ming-Hsuan Yang. Tracking persons-of-interest via unsupervised representation adaptation. *International Journal of Computer Vision*, 128(1):96–120, 2020.

[37] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for image restoration with neural networks. *IEEE Transactions on computational imaging*, 3(1):47–57, 2016.

[38] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.